Static analysis of run-time errors in embedded real-time parallel C programs

Antoine Miné

STATIC ANALYSIS OF RUN-TIME ERRORS IN EMBEDDED REAL-TIME PARALLEL C PROGRAMS

ANTOINE MINÉ

CNRS & École Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France
e-mail address: mine@di.ens.fr

Abstract. We present a static analysis by Abstract Interpretation to check for run-time errors in parallel and multi-threaded C programs. Following our work on Astrée, we focus on embedded critical programs without recursion nor dynamic memory allocation, but extend the analysis to a static set of threads communicating implicitly through a shared memory and explicitly using a finite set of mutual exclusion locks, and scheduled according to a real-time scheduling policy and fixed priorities. Our method is thread-modular. It is based on a slightly modified non-parallel analysis that, when analyzing a thread, applies and enriches an abstract set of thread interferences. An iterator then re-analyzes each thread in turn until interferences stabilize. We prove the soundness of our method with respect to the sequential consistency semantics, but also with respect to a reasonable weakly consistent memory semantics. We also show how to take into account mutual exclusion and thread priorities through a partitioning over an abstraction of the scheduler state. We present preliminary experimental results analyzing an industrial program with our prototype, Thésée, and demonstrate the scalability of our approach.

1. Introduction

Ensuring the safety of critical embedded software is important as a single "bug" can have catastrophic consequences. Previous work on the Astrée analyzer [8] demonstrated that static analysis by Abstract Interpretation could help, when specializing an analyzer to a class of properties and programs — namely in that case, the absence of run-time errors (such as arithmetic and memory errors) on synchronous control / command embedded avionic C software. In this article, we describe ongoing work to achieve similar results for *multi-threaded* and *parallel* embedded C software. Such an extension is demanded by the current trend in critical embedded systems to switch from large numbers of single-program

1998 ACM Subject Classification: D.2.4, F.3.1, F.3.2.

Key words and phrases: Abstract interpretation, parallel programs, run-time errors, static analysis.

This article is an extended version of our article [47] published in the Proceedings of the 20th European Symposium on Programming (ESOP'11).

This work was partially supported by the INRIA project "Abstraction" common to CNRS and ENS in France, and by the project ANR-11-INSE-014 from the French *Agence nationale de la recherche*.

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

processors communicating through a common bus to single-processor multi-threaded applications communicating through a shared memory — for instance, in the context of Integrated Modular Avionics [60]. Analyzing each thread independently with a tool such as Astrée would not be sound and could miss bugs that only appear when threads interact. In this article, we focus on detecting the same kinds of run-time errors as Astrée does, while taking thread communications into account in a sound way, including accesses to the shared memory and synchronization primitives. In particular, we correctly handle the effect of concurrent threads accessing a common variable without enforcing mutual exclusion by synchronization primitives, and we report such accesses — these will be called *data-races* in the rest of the article. However, we ignore other concurrency hazards such as dead-locks, live-locks, and priority inversions, which are considered to be orthogonal issues.

Our method is based on Abstract Interpretation [13], a general theory of the approximation of semantics which allows designing static analyzers that are fully automatic and sound by construction — i.e., consider a superset of all program behaviors. Such analyzers cannot miss any bug in the class of errors they analyze. However, they can cause spurious alarms due to over-approximations, an unfortunate effect we wish to minimize while keeping the analysis efficient.

To achieve scalability, our method is thread-modular and performs a rely-guarantee reasoning, where rely and guarantee conditions are inferred automatically. At its core, it performs a sequential analysis of each thread considering an abstraction of the effects of the other threads, called interferences. Each sequential analysis also collects a new set of interferences generated by the analyzed thread. It then serves as input when analyzing the other threads. Starting from an empty set of interferences, threads are re-analyzed in sequence until a fixpoint of interferences is reached for all threads. Using this scheme, few modifications are required to a sequential analyzer in order to analyze multi-threaded programs. Practical experiments suggest that few thread re-analyses are required in practice, resulting in a scalable analysis. The interferences are considered in a flow-insensitive and non-relational way: they store, for each variable, an abstraction of the set of all values it can hold at any program point of a given thread. Our method is however quite generic in the way individual threads are analyzed. They can be analyzed in a fully or partially flow-sensitive, context-sensitive, path-sensitive, and relational way (as is the case in our prototype).

As we target embedded software, we can safely assume that there is no recursion, dynamic allocation of memory, nor dynamic creation of threads nor locks, which makes the analysis easier. In return, we handle two subtle points. Firstly, we consider a weakly consistent memory model: memory accesses not protected by mutual exclusion (i.e., data-races) may cause behaviors that are not the result of any thread interleaving to appear. The reason is that arbitrary observation by concurrent threads can expose compiler and processor optimizations (such as instruction reordering) that are designed to be transparent on non-parallel programs only. We prove that our semantics is invariant by large classes of widespread program transformations, so that an analysis of the original program is also sound with respect to reasonably compiled and optimized versions. Secondly, we show how to take into account the effect of a real-time scheduler that schedules the threads on a single processor following strict, fixed priorities. According to this scheduling algorithm, which is quite common in the realm of embedded real-time software — e.g., in the real-time thread extension of the POSIX standard [34], or in the ARINC 653 avionic operating system standard [3] — only the unblocked thread of highest priority may run. This ensures

some lock-less mutual exclusion properties that are actually exploited in real-time embedded programs and relied on for their correctness (this includes the industrial application our prototype currently targets). We show how our analysis can take these properties into account, but we also present an analysis that assumes less properties on the scheduler and is thus sound for true multi-processors and non-real-time schedulers. We handle synchronization properties (enforced by either locks or priorities) through a partitioning with respect to an abstraction of the global scheduling state. The partitioning recovers some kind of inter-thread flow-sensitivity that would otherwise be completely abstracted away by the interference abstraction.

The approach presented in this article has been implemented and used at the core of a prototype analyzer named Thésée. It leverages the static analysis techniques developed in Astrée [8] for single-threaded programs, and adds the support for multiple threads. We used Thésée to analyze in 27 h a large (1.7 M lines) multi-threaded industrial embedded C avionic application, which illustrates the scalability of our approach.

Organisation. Our article is organized as follows. First, Sec. 2 presents a classic non-parallel semantics and its static analysis. Then, Sec. 3 extends them to several threads in a shared memory and discusses weakly consistent memory issues. A model of the scheduler and support for locks and priorities are introduced in Sec. 4. Our prototype analyzer, Thésée, is presented in Sec. 5, as well as some experimental results. Finally, Sec. 6 discusses related work, and Sec. 7 concludes and envisions future work.

This article defines many semantics. They are summarized in Fig. 1, using $\subseteq$ to denote the "is less abstract than" relation. We alternate between two kinds of concrete semantics: semantics based on *control paths* ($\mathbb{P}_\pi$, $\mathbb{P}_*$, $\mathbb{P}_\mathcal{H}$), that can model precisely thread interleavings and are also useful to characterize weakly consistent memory models ($\mathbb{P}'_*$, $\mathbb{P}'_\mathcal{H}$), and semantics by *structural induction* on the syntax ($\mathbb{P}$, $\mathbb{P}_\mathcal{I}$, $\mathbb{P}_\mathcal{C}$), that give rise to effective abstract interpreters ($\mathbb{P}^\sharp$, $\mathbb{P}^\sharp_\mathcal{I}$, $\mathbb{P}^\sharp_\mathcal{C}$). Each semantics is presented in its subsection and adds some features to the previous ones, so that the final abstract analysis $\mathbb{P}^\sharp_\mathcal{C}$ presented in Sec. 4.5 should hopefully not appear as too complex nor artificial, but rather as the logical conclusion of a step-by-step construction.

Our analysis has been mentioned first, briefly and informally, in [7, § VI]. We offer here a formal, rigorous treatment by presenting all the semantics fully formally, albeit on an idealised language, and by studying their relationship. The present article is an extended version of [47] and includes a more comprehensive description of the semantics as well as the proof of all theorems, that were omitted in the conference proceedings due to lack of space.

Notations. In this article, we use the theory of *complete lattices*, denoting their partial order, join, and least element respectively as $\sqsubseteq$, $\sqcup$, and $\bot$, possibly with some subscript to indicate which lattice is considered. All the lattices we use are actually constructed by taking the Cartesian product of one or several *powerset* lattices — i.e., $\mathcal{P}(S)$ for some set S — $\sqsubseteq$, $\sqcup$, and $\bot$ are then respectively the set inclusion $\subseteq$, the set union $\cup$, and the empty set $\emptyset$, applied independently to each component. Given a monotonic operator F in a complete lattice, we denote by $\mathit{lfp}\, F$ its least fixpoint — i.e., $F(\mathit{lfp}\, F) = \mathit{lfp}\, F$ and $\forall X : F(X) = X \implies \mathit{lfp}\, F \sqsubseteq X$ — which exists according to Tarski [59, 14]. We denote by $A \to B$ the set of functions from a set A to a set B, and by $A \xrightarrow{\sqcup} B$ the set of

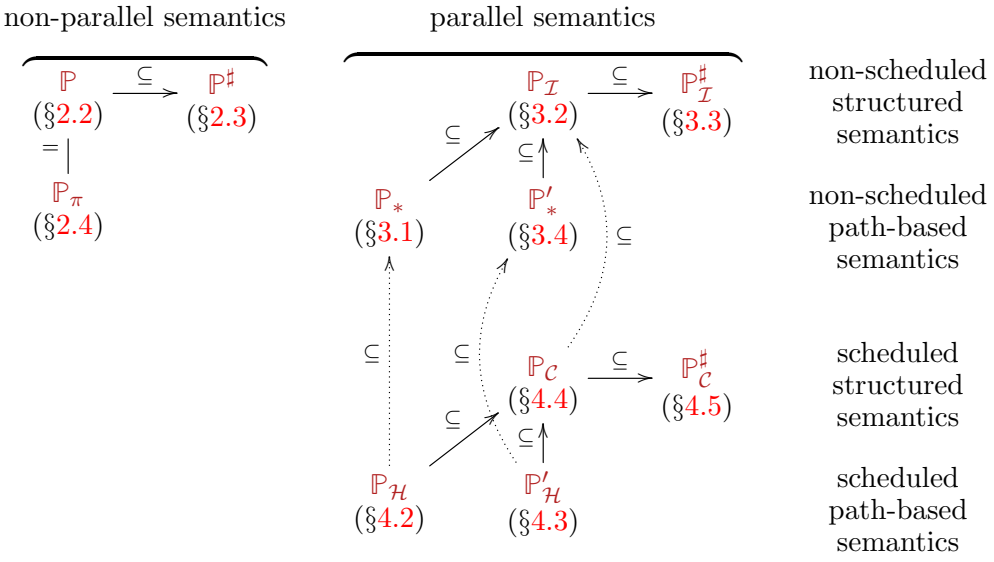


Figure 1: Semantics defined in the article.

complete $\sqcup$−morphisms from a complete lattice A to a complete lattice B, i.e., such that $F(\sqcup_A X) = \bigsqcup_B \{ F(x) \mid x \in X \}$ for any finite or infinite set $X \subseteq A$. Additionally, such a function is monotonic. We use the theory of Abstract Interpretation by Cousot and Cousot and, more precisely, its concretization-based (γ) formalization [16]. We use widenings ($\triangledown$) to ensure termination [17]. The abstract version of a domain, operator, or function is denoted with a $\sharp$ superscript. We use the lambda notation $\lambda x : f(x)$ to denote functions. If f is a function, then $f[x \mapsto v]$ is the function with the same domain as f that maps x to v, and all other elements $y \neq x$ to $f(y)$. Likewise, $f[\forall x \in X : x \mapsto g(x)]$ denotes the function that maps any $x \in X$ to $g(x)$, and other elements $y \notin X$ to $f(y)$. Boldface fonts are used for syntactic elements, such as "**while**" in Fig. 2. Pairs and tuples are bracketed by parentheses, as in $X = (A, B, C)$, and can be deconstructed (matched) with the notation "let $(A, -, C) = X$ in $\cdots$" where the "$-$" symbol denotes irrelevant tuple elements. The notation "let $\forall x \in X : y_x = \cdots$ in $\cdots$" is used to bind a collection of variables $(y_x)_{x \in X}$ at once. Semantic functions are denoted with double brackets, as in $\mathbb{X}[\![y]\!]$, where y is an (optional) syntactic object, and $\mathbb{X}$ denotes the kind of objects ($\mathbb{S}$ for statements, $\mathbb{E}$ for expressions, $\mathbb{P}$ for programs, $\mathbb{\Pi}$ for control paths). The kind of semantics considered (parallel, non-parallel, abstract, etc.) is denoted by subscripts and superscripts over $\mathbb{X}$, as exemplified in Fig. 1. Finally, we use finite words over arbitrary sets, using ϵ and $\cdot$ to denote, respectively, the empty word and word concatenation. The concatenation $\cdot$ is naturally extended to sets of words: $A \cdot B \stackrel{\text{def}}{=} \{ a \cdot b \mid a \in A, b \in B \}$.

2. Non-parallel Programs

This section recalls a classic static analysis by Abstract Interpretation of the run-time errors of *non-parallel* programs, as performed for instance by Astrée [8]. The formalization introduced here will be extended later to parallel programs, and it will be apparent that an analyzer for parallel programs can be constructed by extending an analyzer for non-parallel programs with few changes.

$$
\begin{array}{lcll}
\mathit{stat} & ::= & X \leftarrow \mathit{expr} & \textit{(assignment into } X \in \mathcal{V}\textit{)} \\
 & | & \textbf{if } \mathit{expr} \bowtie 0 \textbf{ then } \mathit{stat} & \textit{(conditional)} \\
 & | & \textbf{while } \mathit{expr} \bowtie 0 \textbf{ do } \mathit{stat} & \textit{(loop)} \\
 & | & \mathit{stat};\ \mathit{stat} & \textit{(sequence)} \\
 & & & \\
\mathit{expr} & ::= & X & \textit{(variable } X \in \mathcal{V}\textit{)} \\
 & | & [c_1, c_2] & \textit{(constant interval, } c_1, c_2 \in \mathbb{R} \cup \{\pm\infty\}\textit{)} \\
 & | & -_\ell\ \mathit{expr} & \textit{(unary operation, } \ell \in \mathcal{L}\textit{)} \\
 & | & \mathit{expr} \diamond_\ell \mathit{expr} & \textit{(binary operation, } \ell \in \mathcal{L}\textit{)} \\
 & & & \\
\bowtie & ::= & = \mid \neq \mid < \mid > \mid \leq \mid \geq & \\
\diamond & ::= & + \mid - \mid \times \mid / &
\end{array}
$$

Figure 2: Syntax of programs.

2.1. **Syntax.** For the sake of exposition, we reason on a vastly simplified programming language. However, the results extend naturally to a realistic language, such as the subset of C excluding recursion and dynamic memory allocation considered in our practical experiments (Sec. 5). We assume a fixed, finite set of variable names $\mathcal{V}$. A program is a single structured statement, denoted *body* $\in$ *stat*. The syntax of statements *stat* and of expressions *expr* is depicted in Fig. 2. Constants are actually constant intervals $[c_1, c_2]$, which return a new arbitrary value between c_1 and c_2 every time the expression is evaluated. This allows modeling non-deterministic expressions, such as inputs from the environment, or stubs for expressions that need not be handled precisely, e.g., $\sin(x)$ could be replaced with $[-1, 1]$. Each unary and binary operator $\diamond_\ell$ is tagged with a syntactic location $\ell \in \mathcal{L}$ and we denote by $\mathcal{L}$ the finite set of all syntactic locations. The output of an analyzer will be the set of locations ℓ with errors — or rather, a superset of them, due to approximations.

For the sake of simplicity, we do not handle procedures. These are handled by inlining in our prototype. We also focus on a single data-type (real numbers in $\mathbb{R}$) and numeric expressions, which are sufficient to provide interesting properties to express, e.g., variable bounds, although in the following we will only discuss proving the absence of division by zero. Handling of realistic data-types (machine integers, floats arrays, structures, pointers, etc.) and more complex properties (such as the absence of numeric and pointer overflow) as done in our prototype is orthogonal, and existing methods apply directly — for instance [7].

2.2. **Concrete Structured Semantics $\mathbb{P}$.** As usual in Abstract Interpretation, we start by providing a concrete semantics, that is, the most precise mathematical expression of program semantics we consider. It should be able to express the properties of interest to us, i.e., which run-time errors can occur — only divisions by zero for the simplified language of Fig. 2. For this, it is sufficient that our concrete semantics tracks numerical invariants. As this problem is undecidable, it will be abstracted in the next section to obtain a sound static analysis.

A program environment $\rho \in \mathcal{E}$ maps each variable to a value, i.e., $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{R}$. The semantics $\mathbb{E}[\![\, e \,]\!]$ of an expression $e \in \mathit{expr}$ takes as input a single environment ρ, and outputs a set of values, in $\mathcal{P}(\mathbb{R})$, and a set of locations of run-time errors, in $\mathcal{P}(\mathcal{L})$. It is defined by structural induction in Fig. 3. Note that an expression can evaluate to one value, several

$$\underline{\mathbb{E}[\![\, e \,]\!] : \mathcal{E} \to (\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\mathcal{L}))}$$
$$\mathbb{E}[\![\, X \,]\!]\rho \stackrel{\text{def}}{=} (\{\, \rho(X) \,\}, \emptyset)$$
$$\mathbb{E}[\![\, [c_1, c_2] \,]\!]\rho \stackrel{\text{def}}{=} (\{\, c \in \mathbb{R} \mid c_1 \leq c \leq c_2 \,\}, \emptyset)$$
$$\mathbb{E}[\![\, -_\ell\ e \,]\!]\rho \stackrel{\text{def}}{=} \text{let } (V, \Omega) = \mathbb{E}[\![\, e \,]\!]\ \rho \text{ in } (\{\, -x \mid x \in V \,\}, \Omega)$$
$$\begin{aligned}\mathbb{E}[\![\, e_1 \diamond_\ell e_2 \,]\!]\rho \stackrel{\text{def}}{=}\ & \text{let } (V_1, \Omega_1) = \mathbb{E}[\![\, e_1 \,]\!]\ \rho \text{ in} \\ & \text{let } (V_2, \Omega_2) = \mathbb{E}[\![\, e_2 \,]\!]\ \rho \text{ in} \\ & (\{\, x_1 \diamond x_2 \mid x_1 \in V_1,\ x_2 \in V_2,\ \diamond \neq / \vee x_2 \neq 0 \,\}, \\ & \ \Omega_1 \cup \Omega_2 \cup \{\, \ell \mid \diamond = / \wedge 0 \in V_2 \,\})\end{aligned}$$
where $\diamond \in \{\, +, -, \times, / \,\}$

Figure 3: Concrete semantics of expressions.

values (due to non-determinism in $[c_1, c_2]$) or no value at all (in the case of a division by zero).

To define the semantics of statements, we consider as semantic domain the complete lattice:

$$\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\mathcal{L}) \tag{2.1}$$

with partial order $\sqsubseteq$ defined as the pairwise set inclusion: $(A, B) \sqsubseteq (A', B') \stackrel{\text{def}}{\iff} A \subseteq A' \wedge B \subseteq B'$. We denote by $\sqcup$ the associated join, i.e., pairwise set union. The *structured* semantics $\mathbb{S}[\![\, s \,]\!]$ of a statement s is a morphism in $\mathcal{D}$ that, given a set of environments R and errors Ω before a statement s, returns the reachable environments after s, as well as Ω enriched with the errors encountered during the execution of s. It is defined by structural induction in Fig. 4. We introduce the new statements $e \bowtie 0?$ (where $\bowtie \in \{\, =, \neq, <, >, \leq, \geq \,\}$ is a comparison operator) which we call "guards." These statements do not appear stand-alone in programs, but are useful to factor the semantic definition of conditionals and loops (they are similar to the guards used in Dijkstra's Guarded Commands [25]). Guards will also prove useful to define control paths in Sec. 2.4. Guards filter their argument and keep only those environments where the expression e evaluates to a set containing a value v satisfying $v \bowtie 0$. The symbol $\not\bowtie$ denotes the negation of $\bowtie$, i.e., the negation of $=, \neq, <, >, \leq, \geq$ is, respectively, $\neq, =, \geq, \leq, >, <$. Finally, the semantics of loops computes a loop invariant using the least fixpoint operator *lfp*. The fact that such fixpoints exist, and the related fact that the semantic functions are complete $\sqcup-$morphisms, i.e., $\mathbb{S}[\![\, s \,]\!](\sqcup_{i \in I} X_i) = \sqcup_{i \in I}\ \mathbb{S}[\![\, s \,]\!] X_i$, is stated in the following theorem:

Theorem 2.1. *$\forall s \in$ stat : $\mathbb{S}[\![\, s \,]\!]$ is well defined and a complete $\sqcup-$morphism.*

Proof. In Appendix A.1. □

We can now define the concrete structured semantics of the program as follows:

$$\mathbb{P} \stackrel{\text{def}}{=} \Omega, \text{ where } (-, \Omega) = \mathbb{S}[\![\ \mathit{body}\ ]\!](\mathcal{E}_0, \emptyset) \tag{2.2}$$

where $\mathcal{E}_0 \subseteq \mathcal{E}$ is a set of initial environments. We can choose, for instance, $\mathcal{E}_0 = \mathcal{E}$ or $\mathcal{E}_0 \stackrel{\text{def}}{=} \{\, \lambda X \in \mathcal{V} : 0 \,\}$. Note that all run-time errors are collected while traversing the program structure; they are never discarded and all of them eventually reach the end of *body*, and so, appear in $\mathbb{P}$, even if $\mathbb{S}[\![\ \mathit{body}\ ]\!](\mathcal{E}_0, \emptyset)$ outputs an empty set of environments. Our program semantics thus observes the set of run-time errors that can appear in any execution

$$\underline{\mathbb{S}[\![\, s \,]\!] : \mathcal{D} \xrightarrow{\sqcup} \mathcal{D}}$$

$$\mathbb{S}[\![\, X \leftarrow e \,]\!](R,\Omega) \stackrel{\text{def}}{=} (\emptyset,\Omega) \sqcup \bigsqcup_{\rho\in R} \text{let } (V,\Omega') = \mathbb{E}[\![\, e \,]\!]\,\rho \text{ in } (\{\, \rho[X \mapsto v] \mid v \in V \,\}, \Omega')$$

$$\mathbb{S}[\![\, e \bowtie 0? \,]\!](R,\Omega) \stackrel{\text{def}}{=} (\emptyset,\Omega) \sqcup \bigsqcup_{\rho\in R} \text{let } (V,\Omega') = \mathbb{E}[\![\, e \,]\!]\,\rho \text{ in } (\{\, \rho \mid \exists v \in V : v \bowtie 0 \,\}, \Omega')$$

$$\mathbb{S}[\![\, s_1;\ s_2 \,]\!](R,\Omega) \stackrel{\text{def}}{=} (\mathbb{S}[\![\, s_2 \,]\!] \circ \mathbb{S}[\![\, s_1 \,]\!])(R,\Omega)$$

$$\mathbb{S}[\![\, \textbf{if } e \bowtie 0 \textbf{ then } s \,]\!](R,\Omega) \stackrel{\text{def}}{=} (\mathbb{S}[\![\, s \,]\!] \circ \mathbb{S}[\![\, e \bowtie 0? \,]\!])(R,\Omega) \sqcup \mathbb{S}[\![\, e \not\bowtie 0? \,]\!](R,\Omega)$$

$$\mathbb{S}[\![\, \textbf{while } e \bowtie 0 \textbf{ do } s \,]\!](R,\Omega) \stackrel{\text{def}}{=} \mathbb{S}[\![\, e \not\bowtie 0? \,]\!]\,(\mathit{lfp}\ \lambda X : (R,\Omega) \sqcup (\mathbb{S}[\![\, s \,]\!] \circ \mathbb{S}[\![\, e \bowtie 0? \,]\!])X)$$

where $\bowtie \in \{=, \neq, <, >, \leq, \geq\}$

Figure 4: Structured concrete semantics of statements.

starting at the beginning of *body* in an initial environment. This includes errors occurring in executions that loop forever (such as infinite reactive loops in control / command software) or that halt before the end of *body*.

2.3. **Abstract Structured Semantics $\mathbb{P}^\sharp$.** The semantics $\mathbb{P}$ is not computable as it involves least fixpoints in an infinite-height domain $\mathcal{D}$, and not all elements in $\mathcal{D}$ are representable in a computer as $\mathcal{D}$ is uncountable. Even if we restricted variable values to a more realistic, large but finite, subset — such as machine integers or floats — naive computation in $\mathcal{D}$ would be unpractical. An effective analysis will instead compute an abstract semantics over-approximating the concrete one.

The abstract semantics is parametrized by the choice of an abstract domain of environments obeying the signature presented in Fig. 5. It comprises a set $\mathcal{E}^\sharp$ of computer-representable abstract environments, with a partial order $\subseteq^\sharp_\mathcal{E}$ (denoting abstract entailment) and an abstract environment $\mathcal{E}^\sharp_0 \in \mathcal{E}^\sharp$ representing initial environments. Each abstract environment represents a set of concrete environments through a monotonic concretization function $\gamma_\mathcal{E} : \mathcal{E}^\sharp \to \mathcal{P}(\mathcal{E})$. We also require an effective abstract version $\cup^\sharp_\mathcal{E}$ of the set union $\cup$, as well as effective abstract versions $\mathbb{S}^\sharp[\![\, s \,]\!]$ of the semantic operators $\mathbb{S}[\![\, s \,]\!]$ for assignment and guard statements. Only environment sets are abstracted, while error sets are represented explicitly, so that the actual abstract semantic domain for $\mathbb{S}^\sharp[\![\, s \,]\!]$ is $\mathcal{D}^\sharp \stackrel{\text{def}}{=} \mathcal{E}^\sharp \times \mathcal{P}(\mathcal{L})$, with concretization γ defined in Fig. 5. Figure 5 also presents the soundness conditions that state that an abstract operator outputs a superset of the environments and error locations returned by its concrete version. Finally, when $\mathcal{E}^\sharp$ has infinite strictly increasing chains, we require a widening operator $\triangledown_\mathcal{E}$, i.e., a sound abstraction of the join $\cup$ with a termination guarantee to ensure the convergence of abstract fixpoint computations in finite time. There exist many abstract domains $\mathcal{E}^\sharp$, for instance the interval domain [13], where an abstract environment in $\mathcal{E}^\sharp$ associates an interval to each variable, the octagon domain [46], where an abstract environment in $\mathcal{E}^\sharp$ is a conjunction of constraints of the form $\pm X \pm Y \leq c$ with $X, Y \in \mathcal{V}$, $c \in \mathbb{R}$, or the polyhedra domain [21], where an abstract environment in $\mathcal{E}^\sharp$ is a convex, closed (possibly unbounded) polyhedron.

In the following, we will refer to assignments and guards collectively as *primitive statements*. Their abstract semantics $\mathbb{S}^\sharp[\![\, s \,]\!]$ in $\mathcal{D}^\sharp$ depends on the choice of abstract domain; we assume it is provided as part of the abstract domain definition and do not discuss it. By contrast, the semantics of non-primitive statements can be derived in a generic way,

$\mathcal{E}^\sharp$ *(set of abstract environments)*

$\gamma_\mathcal{E} : \mathcal{E}^\sharp \to \mathcal{P}(\mathcal{E})$ *(concretization)*

$\bot^\sharp_\mathcal{E} \in \mathcal{E}^\sharp$ *(empty abstract environment)*
s.t. $\gamma_\mathcal{E}(\bot^\sharp_\mathcal{E}) = \emptyset$

$\mathcal{E}^\sharp_0 \in \mathcal{E}^\sharp$ *(initial abstract environment)*
s.t. $\gamma_\mathcal{E}(\mathcal{E}^\sharp_0) \supseteq \mathcal{E}_0$

$\subseteq^\sharp_\mathcal{E}: (\mathcal{E}^\sharp \times \mathcal{E}^\sharp) \to \{ \mathit{true}, \mathit{false} \}$ *(abstract entailment)*
s.t. $X^\sharp \subseteq^\sharp_\mathcal{E} Y^\sharp \Longrightarrow \gamma_\mathcal{E}(X^\sharp) \subseteq \gamma_\mathcal{E}(Y^\sharp)$

$\cup^\sharp_\mathcal{E} : (\mathcal{E}^\sharp \times \mathcal{E}^\sharp) \to \mathcal{E}^\sharp$ *(abstract join)*
s.t. $\gamma_\mathcal{E}(X^\sharp \cup^\sharp_\mathcal{E} Y^\sharp) \supseteq \gamma_\mathcal{E}(X^\sharp) \cup \gamma_\mathcal{E}(Y^\sharp)$

$\triangledown_\mathcal{E} : (\mathcal{E}^\sharp \times \mathcal{E}^\sharp) \to \mathcal{E}^\sharp$ *(widening)*
s.t. $\gamma_\mathcal{E}(X^\sharp \triangledown_\mathcal{E} Y^\sharp) \supseteq \gamma_\mathcal{E}(X^\sharp) \cup \gamma_\mathcal{E}(Y^\sharp)$
and $\forall (Y^\sharp_i)_{i\in\mathbb{N}}$: the sequence $X^\sharp_0 = Y^\sharp_0$, $X^\sharp_{i+1} = X^\sharp_i \triangledown_\mathcal{E} Y^\sharp_{i+1}$
reaches a fixpoint $X^\sharp_k = X^\sharp_{k+1}$ for some $k \in \mathbb{N}$

$\mathcal{D}^\sharp \stackrel{\text{def}}{=} \mathcal{E}^\sharp \times \mathcal{P}(\mathcal{L})$ *(abstraction of $\mathcal{D}$)*

$\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ *(concretization for $\mathcal{D}^\sharp$)*
s.t. $\gamma(R^\sharp, \Omega) \stackrel{\text{def}}{=} (\gamma_\mathcal{E}(R^\sharp), \Omega)$

$\mathbb{S}^\sharp[\![s]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$
s.t. $\forall s \in \{ X \leftarrow e,\ e \bowtie 0? \} : (\mathbb{S}[\![s]\!] \circ \gamma)(R^\sharp, \Omega) \sqsubseteq (\gamma \circ \mathbb{S}^\sharp[\![s]\!])(R^\sharp, \Omega)$

Figure 5: Abstract domain signature, and soundness and termination conditions.

as presented in Fig. 6. Note the similarity between these definitions and the concrete semantics of Fig. 4, except for the semantics of loops that uses additionally a widening operator $\triangledown$ derived from $\triangledown_\mathcal{E}$. The termination guarantee of the widening ensures that, given any (not necessarily monotonic) function $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$, the sequence $X^\sharp_0 \stackrel{\text{def}}{=} (\bot^\sharp_\mathcal{E}, \emptyset)$, $X^\sharp_{i+1} \stackrel{\text{def}}{=} X^\sharp_i \triangledown F^\sharp(X^\sharp_i)$ reaches a fixpoint $X^\sharp_k = X^\sharp_{k+1}$ in finite time $k \in \mathbb{N}$. We denote this limit by $\mathit{lim}\, \lambda X^\sharp : X^\sharp \triangledown F^\sharp(X^\sharp)$. Note that, due to widening, the semantics of a loop is generally not a join morphism, and even not monotonic [17], even if the semantics of the loop body is. Hence, there would be little benefit in imposing that the semantics of primitive statements provided with $\mathcal{D}^\sharp$ is monotonic, and we do not impose it in Fig. 5. Note also that $\mathit{lim}\, F^\sharp$ may not be the least fixpoint of $F^\sharp$ (in fact, such a least fixpoint may not even exist).

The abstract semantics of a program can then be defined, similarly to (2.2), as:

$$\mathbb{P}^\sharp \stackrel{\text{def}}{=} \Omega, \text{ where } (-, \Omega) = \mathbb{S}^\sharp[\![\mathit{body}]\!](\mathcal{E}^\sharp_0, \emptyset) \ .$$

The following theorem states the soundness of the abstract semantics:

Theorem 2.2. $\mathbb{P} \subseteq \mathbb{P}^\sharp$.

Proof. In Appendix A.2. □

The resulting analysis is flow-sensitive. It is relational whenever $\mathcal{E}^\sharp$ is — e.g., with octagons [46]. The iterator follows, in the terminology of [10], a recursive iteration strategy.

$$\underline{\mathbb{S}^\sharp[\![\, s \,]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp}$$

$$\mathbb{S}^\sharp[\![\, s_1;\ s_2 \,]\!](R^\sharp, \Omega) \stackrel{\text{def}}{=} (\mathbb{S}^\sharp[\![\, s_2 \,]\!] \circ \mathbb{S}^\sharp[\![\, s_1 \,]\!])(R^\sharp, \Omega)$$

$$\mathbb{S}^\sharp[\![\, \textbf{if}\ e \bowtie 0\ \textbf{then}\ s \,]\!](R^\sharp, \Omega) \stackrel{\text{def}}{=} (\mathbb{S}^\sharp[\![\, s \,]\!] \circ \mathbb{S}^\sharp[\![\, e \bowtie 0? \,]\!])(R^\sharp, \Omega)\ \cup^\sharp\ \mathbb{S}^\sharp[\![\, e \not\bowtie 0? \,]\!](R^\sharp, \Omega)$$

$$\mathbb{S}^\sharp[\![\, \textbf{while}\ e \bowtie 0\ \textbf{do}\ s \,]\!](R^\sharp, \Omega) \stackrel{\text{def}}{=} \mathbb{S}^\sharp[\![\, e \not\bowtie 0? \,]\!](\mathit{lim}\ \lambda X^\sharp : X^\sharp \triangledown ((R^\sharp, \Omega)\ \cup^\sharp\ (\mathbb{S}^\sharp[\![\, s \,]\!] \circ \mathbb{S}^\sharp[\![\, e \bowtie 0? \,]\!])X^\sharp))$$

where:

$$(R_1^\sharp, \Omega_1)\ \cup^\sharp\ (R_2^\sharp, \Omega_2) \stackrel{\text{def}}{=} (R_1^\sharp \cup_{\mathcal{E}}^\sharp R_2^\sharp,\ \Omega_1 \cup \Omega_2)$$

$$(R_1^\sharp, \Omega_1) \triangledown (R_2^\sharp, \Omega_2) \stackrel{\text{def}}{=} (R_1^\sharp \triangledown_{\mathcal{E}} R_2^\sharp,\ \Omega_1 \cup \Omega_2)$$

Figure 6: Derived abstract functions for non-primitive statements.

The advantage of this strategy is its efficient use of memory: few abstract elements need to be kept in memory during the analysis. Indeed, apart from the current abstract environment, a clever implementation of Fig. 6 exploiting tail recursion would only need to keep one extra environment per **if** $e \bowtie 0$ **then** s statement — to remember the $(R^\sharp, \Omega)$ argument while evaluating s — and two environments per **while** $e \bowtie 0$ **do** s statement — one for $(R^\sharp, \Omega)$ and one for the accumulator $X^\sharp$ — in the call stack of the abstract interpreter function $\mathbb{S}^\sharp$. Thus, the maximum memory consumption is a function of the maximum nesting of conditionals and loops in the analyzed program, which is generally low. This efficiency is key to analyze large programs, as demonstrated by Astrée [8].

2.4. **Concrete Path-Based Semantics $\mathbb{P}_\pi$.** The structured semantics of Sec. 2.2 is defined as an interpretation of the program by induction on its syntactic structure, which can be conveniently transformed into a static analyzer, as shown in Sec. 2.3. Unfortunately, the execution of a parallel program does not follow such a simple syntactic structure; it is rather defined as an interleaving of control paths from distinct threads (Sec. 3.1). Before considering parallel programs, we start by proposing in this section an alternate concrete semantics of non-parallel programs based on control paths. While its definition is different from the structured semantics of Sec. 2.2, its output is equivalent.

A *control path* p is any finite sequence of primitive statements, among $X \leftarrow e$, $e \bowtie 0?$. We denote by Π the set of all control paths. Given a statement s, the set of control paths it spawns $\pi(s) \subseteq \Pi$ is defined by structural induction as follows:

$$\begin{aligned}
&\pi(X \leftarrow e) \stackrel{\text{def}}{=} \{\, X \leftarrow e \,\} \\
&\pi(s_1;\ s_2) \stackrel{\text{def}}{=} \pi(s_1) \cdot \pi(s_2) \\
&\pi(\textbf{if}\ e \bowtie 0\ \textbf{then}\ s) \stackrel{\text{def}}{=} (\{\, e \bowtie 0? \,\} \cdot \pi(s)) \cup \{\, e \not\bowtie 0? \,\} \\
&\pi(\textbf{while}\ e \bowtie 0\ \textbf{do}\ s) \stackrel{\text{def}}{=} (\mathit{lfp}\ \lambda X : \{\epsilon\} \cup (X \cdot \{\, e \bowtie 0? \,\} \cdot \pi(s))) \cdot \{\, e \not\bowtie 0? \,\}
\end{aligned} \tag{2.3}$$

where ϵ denotes then empty path, and $\cdot$ denotes path concatenation, naturally extended to sets of paths. A primitive statement spawns a singleton path of length one, while a conditional spawns two sets of paths — one set where the **then** branch is taken, and one where it is not taken — and loops spawn an infinite number of paths — corresponding to all possible unrollings. Although $\pi(s)$ is infinite whenever s contains a loop, it is possible that

many control paths in $\pi(s)$ are actually infeasible, i.e., have no corresponding execution. In particular, even if a loop s is always bounded, $\pi(s)$ contains unrollings of arbitrary length.

We can now define the semantics $\mathbb{\Pi}[\![P]\!] \in \mathcal{D} \xrightarrow{\sqcup} \mathcal{D}$ of a set of paths $P \subseteq \Pi$ as follows, reusing the semantics of primitive statements from Fig. 4 and the pairwise join $\sqcup$ on sets of environments and errors:

$$\mathbb{\Pi}[\![P]\!]\,(R,\Omega) \overset{\text{def}}{=} \bigsqcup \{\, (\mathbb{S}[\![s_n]\!] \circ \cdots \circ \mathbb{S}[\![s_1]\!])(R,\Omega) \mid s_1 \cdot \ldots \cdot s_n \in P \,\} \ . \tag{2.4}$$

The path-based semantics of a program is then:

$$\mathbb{P}_\pi \overset{\text{def}}{=} \Omega, \text{ where } (-,\Omega) = \mathbb{\Pi}[\![\pi(\mathit{body})]\!](\mathcal{E}_0,\emptyset) \ . \tag{2.5}$$

Note that this semantics is similar to the standard *meet over all paths* solution[1] of data-flow problems — see, e.g., [48, § 2] — but for concrete executions in the infinite-height lattice $\mathcal{D}$. The meet over all paths and maximum fixpoint solutions of data-flow problems are equal for distributive frameworks; similarly, our structured and path-based concrete semantics (based on complete $\sqcup$−morphisms) are equal:

Theorem 2.3. $\forall s \in \mathit{stat} : \mathbb{\Pi}[\![\pi(s)]\!] = \mathbb{S}[\![s]\!]$.

Proof. In Appendix A.3. □

An immediate consequence of this theorem is that $\mathbb{P} = \mathbb{P}_\pi$, hence the two semantics compute, in different ways, the same set of errors.

3. Parallel Programs in a Shared Memory

In this section, we consider several threads that communicate through a shared memory, without any synchronization primitive yet — they will be introduced in Sec. 4. We also discuss here the memory consistency model, and its effect on the semantics and the static analysis.

A program has now a fixed, finite set $\mathcal{T}$ of threads. To each thread $t \in \mathcal{T}$ is associated a statement body $\mathit{body}_t \in \mathit{stat}$. All the variables in $\mathcal{V}$ are shared and can be accessed by all threads.

3.1. Concrete Interleaving Semantics $\mathbb{P}_*$.

The simplest and most natural model of parallel program execution considers all possible interleavings of control paths from all threads. These correspond to *sequentially consistent executions*, as coined by Lamport [39].

A *parallel control path* p is a finite sequence of pairs (s,t), where s is a primitive statement (assignment or guard) and $t \in \mathcal{T}$ is a thread that executes it. We denote by Π_* the set of all parallel control paths. The semantics $\mathbb{\Pi}_*[\![P]\!] \in \mathcal{D} \xrightarrow{\sqcup} \mathcal{D}$ of a set of parallel control paths $P \subseteq \Pi_*$ is defined as in the case of regular control paths (2.4), ignoring thread identifiers:

$$\mathbb{\Pi}_*[\![P]\!]\,(R,\Omega) \overset{\text{def}}{=} \bigsqcup \{\, (\mathbb{S}[\![s_n]\!] \circ \cdots \circ \mathbb{S}[\![s_1]\!])(R,\Omega) \mid (s_1,-) \cdot \ldots \cdot (s_n,-) \in P \,\} \ . \tag{3.1}$$

[1] The lattices used in data-flow analysis and in abstract interpretation are dual: the former use a meet to join paths — hence the expression "meet over all paths" — while we employ a join $\sqcup$. Likewise, the greatest fixpoint solution of a data-flow analysis corresponds to our least fixpoint.

We now denote by $\pi_* \subseteq \Pi_*$ the set of parallel control paths spawned by the whole program. It is defined as:

$$\pi_* \stackrel{\text{def}}{=} \{\, p \in \Pi_* \mid \forall t \in \mathcal{T} : \mathit{proj}_t(p) \in \pi(\mathit{body}_t) \,\} \tag{3.2}$$

where the set $\pi(\mathit{body}_t)$ of regular control paths of a thread is defined in (2.3), and $\mathit{proj}_t(p)$ extracts the maximal sub-path of p on thread t as follows:

$$\begin{array}{l}\mathit{proj}_t((s_1,t_1)\cdot\ldots\cdot(s_n,t_n)) \stackrel{\text{def}}{=} s_{i_1}\cdot\ldots\cdot s_{i_m}\\ \text{such that } \forall j : 1 \leq i_j < i_{j+1} \leq n \ \wedge\ t_{i_j} = t \ \wedge\\ \qquad\qquad \forall k : (k < i_1 \vee k > i_m \vee i_j < k < i_{j+1}) \Longrightarrow t_k \neq t \ .\end{array}$$

The semantics $\mathbb{P}_*$ of a parallel program becomes, similarly to (2.5):

$$\mathbb{P}_* \stackrel{\text{def}}{=} \Omega, \text{ where } (-,\Omega) = \mathbb{\Pi}_*[\![\, \pi_* \,]\!](\mathcal{E}_0, \emptyset) \tag{3.3}$$

i.e., we collect the errors that can appear in any interleaved execution of all the threads, starting in an initial environment.

Because we interleave primitive statements, a thread can only interrupt another one between two primitive statements, and not in the middle of a primitive statement. For instance, in a statement such as $X \leftarrow Y + Y$, no thread can interrupt the current one and change the value of Y between the evaluation of the first and the second Y sub-expression, while it can if the assignment is split into $X \leftarrow Y; X \leftarrow X + Y$. Primitive statements are thus *atomic* in $\mathbb{P}_*$. By contrast, we will present a semantics where primitive statements are not atomic in Sec. 3.4.

3.2. **Concrete Interference Semantics $\mathbb{P}_{\mathcal{I}}$.** Because it reasons on infinite sets of paths, the concrete interleaving semantics from the previous section is not easily amenable to abstraction. In particular, replacing the concrete domain $\mathcal{D}$ in $\mathbb{P}_*$ with an abstract one $\mathcal{D}^\sharp$ (as defined in Sec. 2.3) is not sufficient to obtain an effective and efficient static analyzer as we still have a large or infinite number of paths to analyze separately and join. By contrast, we propose here a (more abstract) concrete semantics that can be expressed by induction on the syntax. It will lead naturally, after further abstraction in Sec. 3.3, to an effective static analysis.

3.2.1. *Thread semantics.* We start by enriching the non-parallel structured semantics of Sec. 2.2 with a notion of interference. We call *interference* a triple $(t, X, v) \in \mathcal{I}$, where $\mathcal{I} \stackrel{\text{def}}{=} \mathcal{T} \times \mathcal{V} \times \mathbb{R}$, indicating that the thread t can set the variable X to the value v. However, it does not say at which program point the assignment is performed, so, it is a flow-insensitive information.

The new semantics of an expression e, denoted $\mathbb{E}_{\mathcal{I}}[\![\, e \,]\!]$, takes as argument the current thread $t \in \mathcal{T}$ and an interference set $I \subseteq \mathcal{I}$ in addition to an environment $\rho \in \mathcal{E}$. It is defined in Fig. 7. The main change with respect to the interference-free semantics $\mathbb{E}[\![\, e \,]\!]$ of Fig. 3 is that, when fetching a variable $X \in \mathcal{V}$, each interference $(t', v, X) \in I$ on X from any other thread $t' \neq t$ is applied. The semantics of constants and operators is not changed, apart from propagating t and I recursively. Note that the choice of evaluating $\mathbb{E}_{\mathcal{I}}[\![\, X \,]\!](t, \rho, I)$ to $\rho(X)$ or to some interference in I, as well as the choice of the interference in I, is non-deterministic. Thus, distinct occurrences of the same variable in an expression may evaluate, in the same environment, to different values.

$$\underline{\mathbb{E}_{\mathcal{I}}[\![\, e \,]\!] : (\mathcal{T} \times \mathcal{E} \times \mathcal{P}(\mathcal{I})) \to (\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\mathcal{L}))}$$

$$\mathbb{E}_{\mathcal{I}}[\![\, X \,]\!](t, \rho, I) \stackrel{\text{def}}{=} (\{\, \rho(X) \,\} \cup \{\, v \mid \exists t' \neq t : (t', X, v) \in I \,\}, \emptyset)$$

$$\mathbb{E}_{\mathcal{I}}[\![\, [c_1, c_2] \,]\!](t, \rho, I) \stackrel{\text{def}}{=} (\{\, c \in \mathbb{R} \mid c_1 \leq c \leq c_2 \,\}, \emptyset)$$

$$\mathbb{E}_{\mathcal{I}}[\![\, -_\ell \; e \,]\!](t, \rho, I) \stackrel{\text{def}}{=} \text{let } (V, \Omega) = \mathbb{E}_{\mathcal{I}}[\![\, e \,]\!](t, \rho, I) \text{ in } (\{\, -x \mid x \in V \,\}, \Omega)$$

$$\begin{array}{l}\mathbb{E}_{\mathcal{I}}[\![\, e_1 \diamond_\ell e_2 \,]\!](t, \rho, I) \stackrel{\text{def}}{=} \\ \quad \text{let } (V_1, \Omega_1) = \mathbb{E}_{\mathcal{I}}[\![\, e_1 \,]\!](t, \rho, I) \text{ in} \\ \quad \text{let } (V_2, \Omega_2) = \mathbb{E}_{\mathcal{I}}[\![\, e_2 \,]\!](t, \rho, I) \text{ in} \\ \quad (\{\, x_1 \diamond x_2 \mid x_1 \in V_1,\, x_2 \in V_2,\, \diamond \neq / \vee x_2 \neq 0 \,\}, \\ \quad\; \Omega_1 \cup \Omega_2 \cup \{\, \ell \mid \diamond = / \wedge 0 \in V_2 \,\}) \\ \text{where } \diamond \in \{\, +, -, \times, / \,\}\end{array}$$

Figure 7: Concrete semantics of expressions with interference.

$$\underline{\mathbb{S}_{\mathcal{I}}[\![\, s, t \,]\!] : \mathcal{D}_{\mathcal{I}} \xrightarrow{\sqcup_{\mathcal{I}}} \mathcal{D}_{\mathcal{I}}}$$

$$\begin{array}{l}\mathbb{S}_{\mathcal{I}}[\![\, X \leftarrow e,\, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} \\ \quad (\emptyset, \Omega, I) \;\sqcup_{\mathcal{I}}\; \bigsqcup_{\rho \in R}^{\mathcal{I}} \text{let } (V, \Omega') = \mathbb{E}_{\mathcal{I}}[\![\, e \,]\!](t, \rho, I) \text{ in} \\ \qquad\qquad\qquad\quad (\{\, \rho[X \mapsto v] \mid v \in V \,\}, \Omega', \{\, (t, X, v) \mid v \in V \,\})\end{array}$$

$$\begin{array}{l}\mathbb{S}_{\mathcal{I}}[\![\, e \bowtie 0?,\, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} \\ \quad (\emptyset, \Omega, I) \;\sqcup_{\mathcal{I}}\; \bigsqcup_{\rho \in R}^{\mathcal{I}} \text{let } (V, \Omega') = \mathbb{E}_{\mathcal{I}}[\![\, e \,]\!](t, \rho, I) \text{ in} \\ \qquad\qquad\qquad\quad (\{\, \rho \mid \exists v \in V : v \bowtie 0 \,\}, \Omega', \emptyset)\end{array}$$

$$\begin{array}{l}\mathbb{S}_{\mathcal{I}}[\![\, \textbf{if } e \bowtie 0 \textbf{ then } s,\, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} \\ \quad (\mathbb{S}_{\mathcal{I}}[\![\, s, t \,]\!] \circ \mathbb{S}_{\mathcal{I}}[\![\, e \bowtie 0?,\, t \,]\!])(R, \Omega, I) \sqcup_{\mathcal{I}} \mathbb{S}_{\mathcal{I}}[\![\, e \not\bowtie 0?,\, t \,]\!](R, \Omega, I)\end{array}$$

$$\begin{array}{l}\mathbb{S}_{\mathcal{I}}[\![\, \textbf{while } e \bowtie 0 \textbf{ do } s,\, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} \\ \quad \mathbb{S}_{\mathcal{I}}[\![\, e \not\bowtie 0?,\, t \,]\!](\mathit{lfp}\, \lambda X : (R, \Omega, I) \sqcup_{\mathcal{I}} (\mathbb{S}_{\mathcal{I}}[\![\, s, t \,]\!] \circ \mathbb{S}_{\mathcal{I}}[\![\, e \bowtie 0?,\, t \,]\!])X)\end{array}$$

$$\mathbb{S}_{\mathcal{I}}[\![\, s_1;\, s_2,\, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} (\mathbb{S}_{\mathcal{I}}[\![\, s_2, t \,]\!] \circ \mathbb{S}_{\mathcal{I}}[\![\, s_1, t \,]\!])(R, \Omega, I)$$

Figure 8: Concrete semantics of statements with interference.

The semantics of a statement s executed by a thread $t \in \mathcal{T}$ is denoted $\mathbb{S}_{\mathcal{I}}[\![\, s, t \,]\!]$. It is presented in Fig. 8. This semantics is enriched with interferences and is thus a complete $\sqcup_{\mathcal{I}}-$morphism in the complete lattice:

$$\mathcal{D}_{\mathcal{I}} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\mathcal{L}) \times \mathcal{P}(\mathcal{I})$$

where the join $\sqcup_{\mathcal{I}}$ is the pairwise set union. The main point of note is the semantics of assignments $X \leftarrow e$. It both uses its interference set argument, passing it to $\mathbb{E}_{\mathcal{I}}[\![\, e \,]\!]$, and enriches it with the interferences generated on the assigned variable X. The semantics of guards simply uses the interference set, while the semantics of conditionals, loops, and sequences is identical to the non-interference one from Fig. 4. The structured semantics of a thread t with interferences I is then $\mathbb{S}_{\mathcal{I}}[\![\, \mathit{body}_t,\, t \,]\!](\mathcal{E}_0, \emptyset, I)$.

3.2.2. *Program semantics.* The semantics $\mathbb{S}_{\mathcal{I}}[\![\, \mathit{body}_t,\, t \,]\!]$ still only analyzes the effect of a single thread t. It assumes *a priori* knowledge of the other threads, through I, and contributes to this knowledge, by enriching I. To solve this dependency and take into account

multiple threads, we iterate the analysis of all threads until interferences stabilize. Thus, the semantics of a multi-threaded program is:

$$\begin{array}{l}\mathbb{P}_{\mathcal{I}} \stackrel{\text{def}}{=} \Omega, \text{ where } (\Omega, -) \stackrel{\text{def}}{=} \\ \quad \mathit{lfp}\, \lambda(\Omega, I) : \bigsqcup_{t\in\mathcal{T}} \text{let } (-, \Omega', I') = \mathbb{S}_{\mathcal{I}}[\![\, \mathit{body}_t,\, t \,]\!]\, (\mathcal{E}_0, \Omega, I) \text{ in } (\Omega', I')\end{array} \tag{3.4}$$

where the join $\sqcup$ is the componentwise set union in the complete lattice $\mathcal{P}(\mathcal{L}) \times \mathcal{P}(\mathcal{I})$.

3.2.3. *Soundness and completeness.* Before linking our interference semantics $\mathbb{P}_{\mathcal{I}}$ (by structural induction) to the interleaving semantics $\mathbb{P}_*$ of Sec. 3.1 (which is path-based), we remark that we can restate the structured interference semantics $\mathbb{S}_{\mathcal{I}}[\![\, \mathit{body}_t,\, t \,]\!]$ of a single thread t in a path-based form, as we did in Sec. 2.4 for non-parallel programs. Indeed, we can replace $\mathbb{S}$ with $\mathbb{S}_{\mathcal{I}}$ in (2.4) and derive a path-based semantics with interference $\mathbb{\Pi}_{\mathcal{I}}[\![\, P,\, t \,]\!] \in \mathcal{D}_{\mathcal{I}} \xrightarrow{\sqcup_{\mathcal{I}}} \mathcal{D}_{\mathcal{I}}$ of a set of (non-parallel) control paths $P \subseteq \Pi$ in a thread t as follows:

$$\mathbb{\Pi}_{\mathcal{I}}[\![\, P, t \,]\!]\, (R, \Omega, I) \stackrel{\text{def}}{=} \bigsqcup\nolimits_{\mathcal{I}} \{\, (\mathbb{S}_{\mathcal{I}}[\![\, s_n, t \,]\!] \circ \cdots \circ \mathbb{S}_{\mathcal{I}}[\![\, s_1, t \,]\!])(R, \Omega, I) \mid s_1 \cdot \ldots \cdot s_n \in P \,\} \ . \tag{3.5}$$

These two forms are equivalent, and Thm. 2.3 naturally becomes:

Theorem 3.1. $\forall t \in \mathcal{T},\, s \in \mathit{stat} : \mathbb{\Pi}_{\mathcal{I}}[\![\, \pi(s),\, t \,]\!] = \mathbb{S}_{\mathcal{I}}[\![\, s,\, t \,]\!]$.

Proof. In Appendix A.4. □

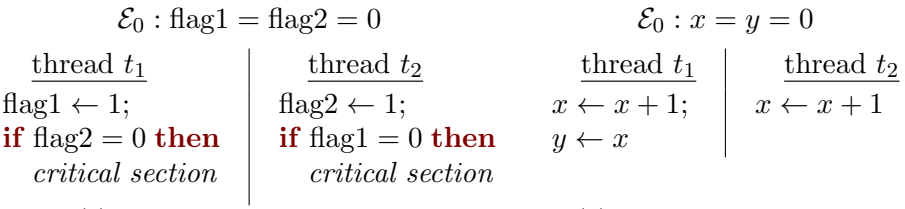
The following theorem then states that the semantics $\mathbb{P}_{\mathcal{I}}$ computed with an interference fixpoint is indeed sound with respect to the interleaving semantics $\mathbb{P}_*$ that interleaves paths from all threads:

Theorem 3.2. $\mathbb{P}_* \subseteq \mathbb{P}_{\mathcal{I}}$.

Proof. In Appendix A.5. □

The equality does not hold in general. Consider, for instance, the program fragment in Fig. 9(a) inspired from Dekker's mutual exclusion algorithm [24]. According to the interleaving semantics, both threads can never be in their critical section simultaneously. The interference semantics, however, does not ensure mutual exclusion. Indeed, it computes the following set of interferences: $\{\, (t_1, \text{flag1}, 1), (t_2, \text{flag2}, 1) \,\}$. Thus, in thread t_1, flag2 evaluates to $\{0, 1\}$. The value 0 comes from the initial state $\mathcal{E}_0$ and the value 1 comes from the interference $(t_2, \text{flag2}, 1)$. Likewise, flag1 evaluates to $\{0, 1\}$ in thread t_2. Thus, both conditions flag1 $= 0$ and flag2 $= 0$ can be simultaneously true. This imprecision is due to the flow-insensitive treatment of interferences. We now present a second example of incompleteness where the loss of precision is amplified by the interference fixpoint. Consider the program in Fig. 9(b) where two threads increment the same zero-initialized variable x. According to the interleaving semantics, either the value 1 or 2 is stored into y. However, in the interference semantics, the interference fixpoint builds a growing set of interferences, up to $\{\, (t, x, i) \mid t \in \{t_1, t_2\},\, i \geq 1 \,\}$, as each thread increments the possible values written by the other thread. Note that the program features no loop and x can thus be incremented only finitely many times (twice), but the interference abstraction is flow-insensitive and forgets how many times an action can be performed. As a consequence, any positive value can be stored into y, instead of only 1 or 2.

Our interference semantics is based on a decomposition of the invariant properties of parallel programs into a local invariant at each thread program point and a global interference invariant. This idea is not new, and complete methods to do so have already been



(a) Mutual exclusion algorithm. (b) Parallel incrementation.

Figure 9: Incompleteness examples for the concrete interference semantics.

proposed. Such methods date back to the works of Owicki, Gries, and Lamport [49, 37, 40] and have been formalized in the framework of Abstract Interpretation by Cousot and Cousot [15]. We would say informally that our interference semantics is an incomplete abstraction of such complete methods, where interferences are abstracted in a flow-insensitive and non-relational way. Our choice to abstract away these information is a deliberate move that eases considerably the construction of an effective and efficient static analyzer, as shown in Sec. 3.3. Another strong incentive is that the interference semantics is compatible with the use of weakly consistent memory models, as shown in Sec. 3.4. Note finally that Sec. 4.4 will present a method to recover a weak form of flow-sensitivity (i.e., mutual exclusion) on interferences, without loosing the efficiency nor the correctness with respect to weak memory models.

3.3. **Abstract Interference Semantics $\mathbb{P}^\sharp_{\mathcal{I}}$.** The concrete interference semantics $\mathbb{P}_{\mathcal{I}}$ introduced in the previous section is defined by structural induction. It can thus be easily abstracted to provide an effective, always-terminating, and sound static analysis.

We assume, as in Sec. 2.3, the existence of an abstract domain $\mathcal{E}^\sharp$ abstracting sets of environments — see Fig. 5. Additionally, we assume the existence of an abstract domain $\mathcal{N}^\sharp$ that abstracts sets of reals, which will be useful to abstract interferences. Its signature is presented in Fig. 10. It is equipped with a concretization $\gamma_{\mathcal{N}} : \mathcal{N}^\sharp \to \mathcal{P}(\mathbb{R})$, a least element $\bot^\sharp_{\mathcal{N}}$, an abstract join $\cup^\sharp_{\mathcal{N}}$ and, if it has strictly increasing infinite chains, a widening $\nabla_{\mathcal{N}}$. We also require two additional functions that will be necessary to communicate information between $\mathcal{E}^\sharp$ and $\mathcal{N}^\sharp$. Firstly, a function $\mathit{get}(X, R^\sharp)$ that extracts from an abstract environment $R^\sharp \in \mathcal{E}^\sharp$ the set of values a variable $X \in \mathcal{V}$ can take, and abstracts this set in $\mathcal{N}^\sharp$. Secondly, a function $\mathit{as\text{-}expr}(V^\sharp)$ able to synthesize a (constant) expression approximating any non-empty abstract value $V^\sharp \in \mathcal{N}^\sharp \setminus \{\bot^\sharp_{\mathcal{N}}\}$. This provides a simple way to use an abstract value from $\mathcal{N}^\sharp$ in functions on abstract environments in $\mathcal{E}^\sharp$. For instance, $\mathbb{S}^\sharp[\![X \leftarrow \mathit{as\text{-}expr}(V^\sharp)]\!](R^\sharp, \Omega)$ non-deterministically sets the variable X in the environments $\gamma_{\mathcal{E}}(R^\sharp)$ to any value in $\gamma_{\mathcal{N}}(V^\sharp)$.

Any non-relational domain on a single variable can be used as $\mathcal{N}^\sharp$. One useful example is the interval domain [13]. In this case, an element in $\mathcal{N}^\sharp$ is either $\bot^\sharp_{\mathcal{N}}$, or a pair consisting of a lower and an upper bound. The function $\mathit{as\text{-}expr}$ is then straightforward because intervals can be directly and exactly represented in the syntax of expressions. Moreover, the function get consists in extracting the range of a variable from an abstract environment $R^\sharp \in \mathcal{E}^\sharp$, an operation which is generally available in the implementations of numerical abstract domains, e.g., in the Apron library [36].

$\mathcal{N}^\sharp$ *(abstract sets of reals)*

$\gamma_{\mathcal{N}} : \mathcal{N}^\sharp \to \mathcal{P}(\mathbb{R})$ *(concretization function)*

$\bot^\sharp_{\mathcal{N}} \in \mathcal{N}^\sharp$ *(abstract empty set)*
s.t. $\gamma_{\mathcal{N}}(\bot^\sharp_{\mathcal{N}}) = \emptyset$

$\cup^\sharp_{\mathcal{N}} : (\mathcal{N}^\sharp \times \mathcal{N}^\sharp) \to \mathcal{N}^\sharp$ *(abstract join)*
s.t. $\gamma_{\mathcal{N}}(V^\sharp \cup^\sharp_{\mathcal{N}} W^\sharp) \supseteq \gamma_{\mathcal{N}}(V^\sharp) \cup \gamma_{\mathcal{N}}(W^\sharp)$

$\triangledown_{\mathcal{N}} : (\mathcal{N}^\sharp \times \mathcal{N}^\sharp) \to \mathcal{N}^\sharp$ *(widening)*
s.t. $\gamma_{\mathcal{N}}(V^\sharp \triangledown_{\mathcal{N}} W^\sharp) \supseteq \gamma_{\mathcal{N}}(V^\sharp) \cup \gamma_{\mathcal{N}}(W^\sharp)$
and $\forall (W^\sharp_i)_{i\in\mathbb{N}}$: the sequence $V^\sharp_0 = W^\sharp_0$, $V^\sharp_{i+1} = V^\sharp_i \triangledown_{\mathcal{N}} W^\sharp_{i+1}$
reaches a fixpoint $V^\sharp_k = V^\sharp_{k+1}$ for some $k \in \mathbb{N}$

$\mathit{get} : (\mathcal{V} \times \mathcal{E}^\sharp) \to \mathcal{N}^\sharp$ *(variable extraction)*
s.t. $\gamma_{\mathcal{N}}(\mathit{get}(X, R^\sharp)) \supseteq \{\, \rho(X) \mid \rho \in \gamma_{\mathcal{E}}(R^\sharp) \,\}$

$\mathit{as\text{-}expr} : (\mathcal{N}^\sharp \setminus \{\bot^\sharp_{\mathcal{N}}\}) \to \mathit{expr}$ *(conversion to expression)*
s.t. $\forall \rho :$ let $(V, -) =$ $\mathbb{E}[\![\ \mathit{as\text{-}expr}(V^\sharp)\ ]\!]\ \rho$ in $V \supseteq \gamma_{\mathcal{N}}(V^\sharp)$

Figure 10: Signature, soundness and termination conditions for a domain $\mathcal{N}^\sharp$ abstracting sets of reals.

$$\mathcal{I}^\sharp \stackrel{\text{def}}{=} (\mathcal{T} \times \mathcal{V}) \to \mathcal{N}^\sharp$$
$$\gamma_{\mathcal{I}} : \mathcal{I}^\sharp \to \mathcal{P}(\mathcal{I})$$
$$\quad \text{s.t. } \gamma_{\mathcal{I}}(I^\sharp) \stackrel{\text{def}}{=} \{\, (t, X, v) \mid t \in \mathcal{T},\ X \in \mathcal{V},\ v \in \gamma_{\mathcal{N}}(I^\sharp(t, X)) \,\}$$
$$\bot^\sharp_{\mathcal{I}} \stackrel{\text{def}}{=} \lambda(t, X) : \bot^\sharp_{\mathcal{N}}$$
$$I^\sharp_1 \cup^\sharp_{\mathcal{I}} I^\sharp_2 \stackrel{\text{def}}{=} \lambda(t, X) : I^\sharp_1(t, X) \cup^\sharp_{\mathcal{N}} I^\sharp_2(t, X)$$
$$I^\sharp_1 \triangledown_{\mathcal{I}} I^\sharp_2 \stackrel{\text{def}}{=} \lambda(t, X) : I^\sharp_1(t, X) \triangledown_{\mathcal{N}} I^\sharp_2(t, X)$$

Figure 11: Abstract domain $\mathcal{I}^\sharp$ of interferences, derived from $\mathcal{N}^\sharp$.

We now show how, given these domains, we can construct an abstraction $\mathbb{P}^\sharp_{\mathcal{I}}$ of $\mathbb{P}_{\mathcal{I}}$. We first construct, using $\mathcal{N}^\sharp$, an abstraction $\mathcal{I}^\sharp$ of interference sets from $\mathcal{P}(\mathcal{I})$, as presented in Fig. 11. It is simply a partitioning of abstract sets of real values with respect to threads and variables: $\mathcal{I}^\sharp \stackrel{\text{def}}{=} (\mathcal{T} \times \mathcal{V}) \to \mathcal{N}^\sharp$, together with pointwise concretization $\gamma_{\mathcal{I}}$, join $\cup^\sharp_{\mathcal{I}}$, and widening $\triangledown_{\mathcal{I}}$. Note that $\mathcal{I}^\sharp$ is not isomorphic to a non-relational domain on a set $\mathcal{T} \times \mathcal{V}$ of variables. Indeed, the former abstracts $(\mathcal{T} \times \mathcal{V}) \to \mathcal{P}(\mathbb{R}) \simeq \mathcal{P}(\mathcal{T} \times \mathcal{V} \times \mathbb{R}) = \mathcal{P}(\mathcal{I})$, while the latter would abstract $\mathcal{P}((\mathcal{T} \times \mathcal{V}) \to \mathbb{R})$. In particular, the former can express abstract states where the value set of some but not all variables is empty, while $\bot^\sharp_{\mathcal{N}}$ elements in the later coalesce to a single element representing $\emptyset$. We then construct an abstraction $\mathcal{D}^\sharp_{\mathcal{I}}$ of the semantic domain $\mathcal{D}_{\mathcal{I}}$, as presented in Fig. 12. An element of $\mathcal{D}^\sharp_{\mathcal{I}}$ is a triple $(R^\sharp, \Omega, I^\sharp)$ composed of an abstraction $R^\sharp \in \mathcal{E}^\sharp$ of environments, a set $\Omega \subseteq \mathcal{L}$ of errors, and an abstraction $I^\sharp \in \mathcal{I}^\sharp$ of interferences. The concretization γ, join $\cup^\sharp$, and widening $\triangledown$ are defined pointwise.

$$\begin{aligned}
&\mathcal{D}^\sharp_\mathcal{I} \stackrel{\text{def}}{=} \mathcal{E}^\sharp \times \mathcal{P}(\mathcal{L}) \times \mathcal{I}^\sharp\\
&\gamma : \mathcal{D}^\sharp_\mathcal{I} \to \mathcal{D}_\mathcal{I}\\
&\qquad \text{s.t. } \gamma(R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} (\gamma_\mathcal{E}(R^\sharp), \Omega, \gamma_\mathcal{I}(I^\sharp))\\
&(R^\sharp_1, \Omega_1, I^\sharp_1) \cup^\sharp (R^\sharp_2, \Omega_2, I^\sharp_2) \stackrel{\text{def}}{=} (R^\sharp_1 \cup^\sharp_\mathcal{E} R^\sharp_2, \Omega_1 \cup \Omega_2, I^\sharp_1 \cup^\sharp_\mathcal{I} I^\sharp_2)\\
&(R^\sharp_1, \Omega_1, I^\sharp_1) \triangledown (R^\sharp_2, \Omega_2, I^\sharp_2) \stackrel{\text{def}}{=} (R^\sharp_1 \triangledown_\mathcal{E} R^\sharp_2, \Omega_1 \cup \Omega_2, I^\sharp_1 \triangledown_\mathcal{I} I^\sharp_2)
\end{aligned}$$

Figure 12: Abstract semantic domain $\mathcal{D}^\sharp_\mathcal{I}$, derived from $\mathcal{E}^\sharp$ and $\mathcal{I}^\sharp$.

The abstract semantics $\mathbb{S}^\sharp_\mathcal{I}[\![s, t]\!]$ of a statement s executed in a thread $t \in \mathcal{T}$ should be a function from $\mathcal{D}^\sharp_\mathcal{I}$ to $\mathcal{D}^\sharp_\mathcal{I}$ obeying the soundness condition:

$$(\mathbb{S}_\mathcal{I}[\![s, t]\!] \circ \gamma)(R^\sharp, \Omega, I^\sharp) \sqsubseteq_\mathcal{I} (\gamma \circ \mathbb{S}^\sharp_\mathcal{I}[\![s, t]\!])(R^\sharp, \Omega, I^\sharp)$$

i.e., the abstract function over-approximates the sets of environments, errors, and interferences. Such a function is defined in a generic way in Fig. 13. The semantics of assignments and guards with interference is defined based on their non-interference semantics $\mathbb{S}^\sharp[\![s]\!]$, provided as part of the abstract domain $\mathcal{E}^\sharp$. In both cases, the expression e to assign or test is first modified to take interferences into account, using the *apply* function. This function takes as arguments a thread $t \in \mathcal{T}$, an abstract environment $R^\sharp \in \mathcal{E}^\sharp$, an abstract interference $I^\sharp \in \mathcal{I}^\sharp$, and an expression e. It first collects, for each variable $Y \in \mathcal{V}$, the relevant interferences $V^\sharp_Y \in \mathcal{N}^\sharp$ from $I^\sharp$, i.e., concerning the variable Y and threads $t' \neq t$. If the interference for Y is empty, $\bot^\sharp_\mathcal{N}$, the occurrences of Y in e are kept unmodified. If it is not empty, then the occurrences of Y are replaced with a constant expression encompassing all the possible values that can be read from Y, from either the interferences or the environments $\gamma_\mathcal{E}(R^\sharp)$. Additionally, the semantics of an assignment $X \leftarrow e$ enriches $I^\sharp$ with new interferences corresponding to the values of X after the assignment. The semantics of non-primitive statements is identical to the interference-free case of Fig. 6.

Finally, an abstraction of the interference fixpoint (3.4) is computed by iteration on abstract interferences, using the widening $\triangledown_\mathcal{I}$ to ensure termination, which provides the abstract semantics $\mathbb{P}^\sharp_\mathcal{I}$ of our program:

$$\begin{aligned}
&\mathbb{P}^\sharp_\mathcal{I} \stackrel{\text{def}}{=} \Omega, \text{ where } (\Omega, -) \stackrel{\text{def}}{=}\\
&\quad \mathit{lim}\, \lambda(\Omega, I^\sharp) : \text{let } \forall t \in \mathcal{T} : (-, \Omega'_t, I^{\sharp\prime}_t) = \mathbb{S}^\sharp_\mathcal{I}[\![\, \mathit{body}_t, t]\!]\, (\mathcal{E}^\sharp_0, \Omega, I^\sharp) \text{ in}\\
&\qquad (\bigcup \{\, \Omega'_t \mid t \in \mathcal{T} \},\ I^\sharp \triangledown_\mathcal{I} \bigcup^\sharp_\mathcal{I} \{\, I^{\sharp\prime}_t \mid t \in \mathcal{T} \})
\end{aligned} \tag{3.6}$$

where $\mathit{lim}\, F^\sharp$ denotes the limit of the iterates of $F^\sharp$ starting from $(\emptyset, \bot^\sharp_\mathcal{I})$. The following theorem states the soundness of the analysis:

Theorem 3.3. $\mathbb{P}_\mathcal{I} \subseteq \mathbb{P}^\sharp_\mathcal{I}$.

Proof. In Appendix A.6. □

The obtained analysis remains flow-sensitive and can be relational within each thread, provided that $\mathcal{E}^\sharp$ is relational. However, interferences are abstracted in a flow-insensitive and non-relational way. This was already the case for the concrete interferences in $\mathbb{P}_\mathcal{I}$ and it is not related to the choice of abstract domains. The analysis is expressed as an outer iteration that completely re-analyzes each thread until the abstract interferences stabilize. Thus, it can be implemented easily on top of an existing non-parallel analyzer. Compared

$$\underline{\mathbb{S}^\sharp_{\mathcal{I}}[\![\, s, t \,]\!] : \mathcal{D}^\sharp_{\mathcal{I}} \to \mathcal{D}^\sharp_{\mathcal{I}}}$$

$$\mathbb{S}^\sharp_{\mathcal{I}}[\![\, X \leftarrow e,\, t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} \text{let } (R^{\sharp\prime}, \Omega') = \mathbb{S}^\sharp[\![\, X \leftarrow \mathit{apply}(t, R^\sharp, I^\sharp, e) \,]\!]\,(R^\sharp, \Omega) \text{ in } (R^{\sharp\prime},\ \Omega',\ I^\sharp[(t, X) \mapsto I^\sharp(t, X) \cup^\sharp_{\mathcal{N}} \mathit{get}(X, R^{\sharp\prime})])$$

$$\mathbb{S}^\sharp_{\mathcal{I}}[\![\, e \bowtie 0?,\, t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} \text{let } (R^{\sharp\prime}, \Omega') = \mathbb{S}^\sharp[\![\, \mathit{apply}(t, R^\sharp, I^\sharp, e) \bowtie 0? \,]\!]\,(R^\sharp, \Omega) \text{ in } (R^{\sharp\prime}, \Omega', I^\sharp)$$

$$\mathbb{S}^\sharp_{\mathcal{I}}[\![\, \textbf{if } e \bowtie 0 \textbf{ then } s,\, t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} (\mathbb{S}^\sharp_{\mathcal{I}}[\![\, s, t \,]\!] \circ \mathbb{S}^\sharp_{\mathcal{I}}[\![\, e \bowtie 0?,\, t \,]\!])(R^\sharp, \Omega, I^\sharp) \ \cup^\sharp\ \mathbb{S}^\sharp_{\mathcal{I}}[\![\, e \not\bowtie 0?,\, t \,]\!](R^\sharp, \Omega, I^\sharp)$$

$$\mathbb{S}^\sharp_{\mathcal{I}}[\![\, \textbf{while } e \bowtie 0 \textbf{ do } s,\, t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} \mathbb{S}^\sharp_{\mathcal{I}}[\![\, e \not\bowtie 0?,\, t \,]\!](\mathit{lim}\ \lambda X^\sharp : X^\sharp \triangledown ((R^\sharp, \Omega, I^\sharp) \ \cup^\sharp\ (\mathbb{S}^\sharp_{\mathcal{I}}[\![\, s, t \,]\!] \circ \mathbb{S}^\sharp_{\mathcal{I}}[\![\, e \bowtie 0?,\, t \,]\!])X^\sharp))$$

$$\mathbb{S}^\sharp_{\mathcal{I}}[\![\, s_1;\, s_2,\, t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} (\mathbb{S}^\sharp_{\mathcal{I}}[\![\, s_2, t \,]\!] \circ \mathbb{S}^\sharp_{\mathcal{I}}[\![\, s_1, t \,]\!])(R^\sharp, \Omega, I^\sharp)$$

where:

$$\mathit{apply}(t, R^\sharp, I^\sharp, e) \stackrel{\text{def}}{=} \text{let } \forall Y \in \mathcal{V} : V^\sharp_Y = \cup^\sharp_{\mathcal{N}}\ \{\, I^\sharp(t', Y) \mid t' \neq t \,\} \text{ in } \text{let } \forall Y \in \mathcal{V} : e_Y = \begin{cases} Y & \text{if } V^\sharp_Y = \bot^\sharp_{\mathcal{N}} \\ \mathit{as\text{-}expr}(V^\sharp_Y \cup^\sharp_{\mathcal{N}} \mathit{get}(Y, R^\sharp)) & \text{if } V^\sharp_Y \neq \bot^\sharp_{\mathcal{N}} \end{cases} \text{ in } e[\forall Y \in \mathcal{V} : Y \mapsto e_Y]$$

Figure 13: Abstract semantics of statements with interference.

to a non-parallel program analysis, the cost is multiplied by the number of outer iterations required to stabilize interferences. Thankfully, our preliminary experimental results suggest that this number remains very low in practice — 5 for our benchmark in Sec. 5. In any case, the overall cost is not related to the (combinatorial) number of possible interleavings, but rather to the amount of abstract interferences $I^\sharp$, i.e., of actual communications between the threads. It is thus always possible to speed up the convergence of interferences or, conversely, improve the precision at the expense of speed, by adapting the widening $\triangledown_{\mathcal{N}}$.

In this article, we focus on analyzing systems composed of a fixed, finite number of threads. The finiteness of $\mathcal{T}$ is necessary for the computation of $\mathbb{P}^\sharp_{\mathcal{I}}$ in (3.6) to be effective. However, it is actually possible to relax this hypothesis and allow an unbounded number of instances of some threads to run in parallel. For this, it is sufficient to consider *self-interferences*, i.e., replace the condition $t' \neq t$ in the definition $\mathbb{E}_{\mathcal{I}}[\![\, X \,]\!](t, \rho, I)$ in Fig. 7 (for the concrete semantics) and $\mathit{apply}(t, R^\sharp, I^\sharp, e)$ in Fig. 13 (for the abstract semantics) with $t' \neq t \vee t \in \mathcal{T}'$, where $\mathcal{T}' \subseteq \mathcal{T}$ denotes the subset of threads that can have several instances. The resulting analysis is necessarily uniform, i.e., it cannot distinguish different instances of the same thread nor express properties about the number of running instances — it is abstracted statically in a domain of two values: "one" ($t \notin \mathcal{T}'$) and "two or more" ($t \in \mathcal{T}'$). In order to analyze actual programs spawning an unbounded number of threads, a non-uniform analysis (such as performed by Feret [26] in the context of the $\pi-$calculus) may be necessary to achieve a sufficient precision, but this is not the purpose of the present article.

3.4. **Weakly Consistent Memory Semantics $\mathbb{P}'_*$.** We now review the various parallel semantics we proposed in the preceding sections and discuss their adequacy to describe actual models of parallel executions.

It appears that our first semantics, the concrete interleaving semantics $\mathbb{P}_*$ of Sec. 3.1, while simple, is not realistic. A first issue is that, as noted by Reynolds in [52], such a semantics requires choosing a level of granularity, i.e., some basic set of operations that are assumed to be atomic and cannot be interrupted by another thread. In our case, we assumed assignments and guards (i.e., primitive statements) to be atomic. In contrast, an actual system may schedule a thread within an assignment and cause, for instance, x to be 1 at the end of the program in Fig. 9(b) instead of the expected value 2. A second issue, noted by Lamport in [38], is that the latency of loads and stores in a shared memory may break the sequential consistency in true multiprocessor systems: threads running on different processors may not always agree on the value of a shared variable. For instance, in the mutual exclusion algorithm of Fig. 9(a), the thread t_2 may still see the value 0 in flag1 even after the thread t_1 has entered its critical section, causing t_2 to also enter its critical section, as the effect of the assignment flag1 $\leftarrow 1$ is propagated asynchronously and takes some time to be acknowledged by t_2. Moreover, Lamport noted in [39] that reordering of independent loads and stores in one thread by the processor can also break sequential consistency — for instance performing the load from flag2 before the store into flag1, instead of after, in the thread t_1 in Fig. 9(a). More recently, it has been observed by Manson et al. [43] that optimizations in modern compilers have the same ill-effect, even on mono-processor systems: program transformations that are perfectly safe on a thread considered in isolation (for instance, reordering the independent assignment flag1 $\leftarrow 1$ and test flag2 $= 0$ in t_1) can cause non-sequentially-consistent behaviors to appear. In this section, we show that the interference semantics correctly handles these issues by proving that it is invariant under a "reasonable" class of program transformations. This is a consequence of its coarse, flow-insensitive and non-relational modeling of thread communications.

Acceptable program transformations of a thread are defined with respect to the path-based semantics $\mathbb{\Pi}$ of Sec. 2.4. A transformation of a thread t is acceptable if it gives rise to a set $\pi'(t) \subseteq \Pi$ of control paths such that every path $p' \in \pi'(t)$ can be obtained from a path $p \in \pi(\mathit{body}_t)$ by a sequence of elementary transformations described below in Def. 3.4. Elementary transformations are denoted $q \rightsquigarrow q'$, where q and q' are sequences of primitive statements. This notation indicates that any occurrence of q in a path of a thread can be replaced with q', whatever the context appearing before and after q. The transformations in Def. 3.4 try to account for widespread compiler and hardware optimizations, but are restricted to transformations that do not generate new errors nor new interferences.[2] This ensures that an interference-based analysis of the original program is sound with respect to the transformed one, which is formalized below in Thm. 3.5.

The elementary transformations of Def. 3.4 require some side-conditions to hold in order to be acceptable. They use the following notions. We say that a variable $X \in \mathcal{V}$ is *fresh* if it does not occur in any thread, and *local* if it occurs only in the currently transformed thread. We denote by $s[e'/e]$ the statement s where some, but not necessarily all, occurrences of the expression e may be changed into e'. The set of variables appearing in the expression e is denoted $\mathit{var}(e)$, while the set of variables modified by the statement s is

[2]The environments at the end of the thread after transformations may be different, but this does not pose a problem as environments are not observable in our semantics: $\mathbb{P}_* \subseteq \mathcal{L}$ (3.3).

$lval(s)$. Thus, $lval(X \leftarrow e) = \{X\}$ while $lval(e \bowtie 0?) = \emptyset$. The predicate $nonblock(e)$ holds if evaluating the expression e cannot block the program — as would, e.g., an expression with a definite run-time error, such as $1/0$ — i.e., $nonblock(e) \stackrel{\text{def}}{\iff} \forall \rho \in \mathcal{E} : V_\rho \neq \emptyset$ where $(V_\rho, -) = \mathbb{E}[\![e]\!]\rho$. We say that e is *deterministic* if, moreover, $\forall \rho \in \mathcal{E} : |V_\rho| = 1$. Finally, $noerror(e)$ holds if evaluating e is always error-free, i.e., $noerror(e) \stackrel{\text{def}}{\iff} \forall \rho \in \mathcal{E} : \Omega_\rho = \emptyset$ where $(-, \Omega_\rho) = \mathbb{E}[\![e]\!]\rho$. We are now ready to state our transformations:

Definition 3.4 (Elementary path transformations)**.**

(1) Redundant store elimination: $X \leftarrow e_1 \cdot X \leftarrow e_2 \rightsquigarrow X \leftarrow e_2$, when $X \notin var(e_2)$ and $nonblock(e_1)$.
(2) Identity store elimination: $X \leftarrow X \rightsquigarrow \epsilon$.
(3) Reordering assignments: $X_1 \leftarrow e_1 \cdot X_2 \leftarrow e_2 \rightsquigarrow X_2 \leftarrow e_2 \cdot X_1 \leftarrow e_1$, when $X_1 \notin var(e_2)$, $X_2 \notin var(e_1)$, $X_1 \neq X_2$, and $nonblock(e_1)$.
(4) Reordering guards: $e_1 \bowtie 0? \cdot e_2 \bowtie' 0? \rightsquigarrow e_2 \bowtie' 0? \cdot e_1 \bowtie 0?$, when $noerror(e_2)$.
(5) Reordering guards before assignments: $X_1 \leftarrow e_1 \cdot e_2 \bowtie 0? \rightsquigarrow e_2 \bowtie 0? \cdot X_1 \leftarrow e_1$, when $X_1 \notin var(e_2)$ and either $nonblock(e_1)$ or $noerror(e_2)$.
(6) Reordering assignments before guards: $e_1 \bowtie 0? \cdot X_2 \leftarrow e_2 \rightsquigarrow X_2 \leftarrow e_2 \cdot e_1 \bowtie 0?$, when $X_2 \notin var(e_1)$, X_2 is local, and $noerror(e_2)$.
(7) Assignment propagation: $X \leftarrow e \cdot s \rightsquigarrow X \leftarrow e \cdot s[e/X]$, when $X \notin var(e)$, $var(e)$ are local, and e is deterministic.
(8) Sub-expression elimination: $s_1 \cdot \ldots \cdot s_n \rightsquigarrow X \leftarrow e \cdot s_1[X/e] \cdot \ldots \cdot s_n[X/e]$, when X is fresh, $\forall i : var(e) \cap lval(s_i) = \emptyset$, and $noerror(e)$.
(9) Expression simplification: $s \rightsquigarrow s[e'/e]$, when $\forall \rho \in \mathcal{E} : \mathbb{E}[\![e]\!]\rho \sqsupseteq \mathbb{E}[\![e']\!]\rho$ and $var(e)$ and $var(e')$ are local.[3]

These simple rules, used in combination, allow modeling large classes of classic program transformations as well as distributed memories. Store latency can be simulated using rules 7 and 3. Breaking a statement into several ones is possible with rules 7 and 8. As a consequence, the rules can expose preemption points within statements, which makes primitive statements no longer atomic. Global optimizations, such as constant propagation and folding, can be achieved using rules 7 and 9. Rules 1–6 allow peephole optimizations. Additionally, transformations that do not change the set of control paths, such as loop unrolling, are naturally supported.

Given the set of transformed control paths $\pi'(t)$ for each thread $t \in \mathcal{T}$, the set of transformed parallel control paths π'_* is defined, similarly to (3.2), as:

$$\pi'_* \stackrel{\text{def}}{=} \{\, p \in \Pi_* \mid \forall t \in \mathcal{T} : proj_t(p) \in \pi'(t) \,\} \tag{3.7}$$

and the semantics $\mathbb{P}'_*$ of the parallel program is, similarly to (3.3):

$$\mathbb{P}'_* \stackrel{\text{def}}{=} \Omega, \text{ where } (-, \Omega) = \mathbb{\Pi}_*[\![\pi'_*]\!](\mathcal{E}_0, \emptyset) \ . \tag{3.8}$$

Any original thread $\pi(body_t)$ being a special case of transformed thread $\pi'(t)$ (considering the identity transformation), we have $\mathbb{P}_* \subseteq \mathbb{P}'_*$. The following theorem extends Thm. 3.2 to transformed programs:

Theorem 3.5. $\mathbb{P}'_* \subseteq \mathbb{P}_{\mathcal{I}}$.

[3]The original expression simplification rule from [47] required a much stronger side-condition: $\mathbb{E}_{\mathcal{I}}[\![e]\!](t, \rho, I) \sqsupseteq \mathbb{E}_{\mathcal{I}}[\![e']\!](t, \rho, I)$ for all ρ and I, which actually implied that e and e' were variable-free. We propose here a more permissive side-condition allowing local variables to appear in e and e'.

Proof. In Appendix A.7. □

An immediate consequence of Thms. 3.3 and 3.5 is the soundness of the abstract semantics $\mathbb{P}^\sharp_{\mathcal{I}}$ with respect to the concrete semantics of the transformed program $\mathbb{P}'_*$, i.e., $\mathbb{P}'_* \subseteq \mathbb{P}^\sharp_{\mathcal{I}}$. Note that, in general, $\mathbb{P}'_* \neq \mathbb{P}_{\mathcal{I}}$. The two semantics may coincide, as for instance in the program of Fig. 9(a). In that case: $\mathbb{P}_* \subsetneq \mathbb{P}'_* = \mathbb{P}_{\mathcal{I}}$. However, in the case of Fig. 9(b), y can take any positive value according to the interference semantics $\mathbb{P}_{\mathcal{I}}$ (as explained in Sec. 3.2), while the interleaving semantics after program transformation $\mathbb{P}'_*$ only allows the values 1 and 2; we have $\mathbb{P}_* = \mathbb{P}'_* \subsetneq \mathbb{P}_{\mathcal{I}}$.

Theorem 3.5 holds for our "reasonable" collection of program transformations, but may not hold when considering other, "unreasonable" ones. For instance, in Fig. 9(a), flag1 $\leftarrow 1$ should not be replaced by a misguided prefetching optimizer with flag1 $\leftarrow 42$; flag1 $\leftarrow 1$ in thread t_1. This would create a spurious interference causing the value 42 to be possibly seen by thread t_2. If there is no other reason for t_2 to see the value 42, such as a previous or future assignment of 42 into flag1 by t_1, it would create an execution outside those considered by the interference semantics and invalidate Thm. 3.5. Such "out-of-thin-air" values are explicitly forbidden by the Java semantics as described by Manson et al. [43]. See also [55] for an in-depth discussion of out-of-thin-air values. Another example of invalid transformation is the reordering of assignments $X_1 \leftarrow e_1 \cdot X_2 \leftarrow e_2 \rightsquigarrow X_2 \leftarrow e_2 \cdot X_1 \leftarrow e_1$ when e_1 may block the program, e.g., due to a division by zero $X_1 \leftarrow 1/0$. Indeed, the transformed program could expose errors in e_2 that cannot occur in the original program because they are masked by the previous error in $X_1 \leftarrow e_1$. This case is explicitly forbidden by the $\mathit{nonblock}(e_1)$ side condition in Def. 3.4.(3). The proof in Appendix A.7 contains more examples of transformations that become invalid when side-conditions are not respected.

Definition 3.4 is not exhaustive. It could be extended with other "reasonable" transformations, and some restrictive side-conditions might be relaxed in future work without breaking Thm. 3.5. It is also possible to enrich Def. 3.4 with new transformations that do not respect Thm. 3.5 as is, and then adapt the interference semantics to retrieve a similar theorem. For instance, we could allow speculative stores of some special value, such as zero, which only requires adding an interference $(t, X, 0)$ for each thread t and each variable X modified by t. As another example, we could consider some memory writes to be non-atomic, such as 64-bit writes on 32-bit computers, which requires adding interferences that expose partially assigned values.

Finally, it would be tempting to, dually, reduce the number of allowed program transformations, and enforce a stronger memory consistency. For instance, we could replace Def. 3.4 with a model of an actual multiprocessor, such as the intel x86 architecture model proposed by Sewell et al. in [57], which is far less permissive and thus ensures many more properties. We would obtain a more precise interleaving semantics $\mathbb{P}'_*$, closer to the sequentially consistent one $\mathbb{P}_*$. However, this would not mechanically improve the result of our static analysis $\mathbb{P}^\sharp_{\mathcal{I}}$, as it is actually an abstraction of the concrete interference semantics $\mathbb{P}_{\mathcal{I}}$, itself an incomplete abstraction of $\mathbb{P}_*$. Our choice of an interference semantics was not initially motivated by the modeling of weakly consistent memories (although this is an important side effect), but rather by the construction of an effective and efficient static analyzer. Effectively translating a refinement of the memory model at the level of an interference-based analysis without sacrificing the efficiency remains a challenging future work.

4. Multi-threaded Programs With a Real-Time Scheduler

We now extend the language and semantics of the preceding section with explicit synchronization primitives. These can be used to enforce mutual exclusion and construct critical sections, avoiding the pitfalls of weakly consistent memories. We also extend the semantics with a real-time scheduler taking thread priorities into account, which provides an alternate way of implementing synchronization.

4.1. Priorities and Synchronization Primitives.

We first describe the syntactic additions to our language and introduce informally the change in semantics.

We denote by $\mathcal{M}$ a finite, fixed set of mutual exclusion locks, so-called *mutexes*. The original language of Fig. 2 is enriched with primitives to control mutexes and scheduling as follows:

$$\begin{array}{rcll} \mathit{stat} & ::= & \textbf{lock}(m) & \textit{(mutex locking, } m \in \mathcal{M}\textit{)} \\ & | & \textbf{unlock}(m) & \textit{(mutex unlocking, } m \in \mathcal{M}\textit{)} \\ & | & X \leftarrow \textbf{islocked}(m) & \textit{(mutex testing, } X \in \mathcal{V}\textit{, } m \in \mathcal{M}\textit{)} \\ & | & \textbf{yield} & \textit{(thread pause)} \end{array} \tag{4.1}$$

The primitives **lock**(m) and **unlock**(m) respectively acquire and release the mutex $m \in \mathcal{M}$. The primitive $X \leftarrow$ **islocked**(m) is used to test the status of the mutex m: it stores 1 into X if m is acquired by some thread, and 0 if it is free. The primitive **yield** is used to voluntarily relinquish the control to the scheduler. The definition of control paths $\pi(s)$ from (2.3) is extended by stating that $\pi(s) \stackrel{\text{def}}{=} \{s\}$ for these statements, i.e., they are primitive statements. We also assume that threads have fixed, distinct priorities. As only the ordering of priorities is significant, we denote threads in $\mathcal{T}$ simply by integers ranging from 1 to $|\mathcal{T}|$, being understood that thread t has a strictly higher priority than thread t' when $t > t'$.

To keep our semantics simple, we assume that acquiring a mutex for a thread already owning it is a no-op, as well as releasing a mutex it does not hold. Our primitive mutexes can serve as the basis to implement more complex ones found in actual implementations. For instance, mutexes that generate a run-time error or return an error code when locked twice by the same thread can be implemented using an extra program variable for each mutex / thread pair that stores whether the thread has already locked that mutex. Likewise, recursive mutexes can be implemented by making these variables count the number of times each thread has locked each mutex. Finally, locking with a timeout can be modeled as a non-deterministic conditional that either locks the mutex, or yields and returns an error code.

Our scheduling model is that of real-time processes, used noticeably in embedded systems. Example operating systems using this model include those obeying the ARINC 653 standard [3] (used in avionics), as well as the real-time extension of the POSIX threads standard [34]. Hard guarantees about the execution time of services, although an important feature of real-time systems, are not the purpose of this article as we abstract physical time away. We are interested in another feature: the strict interpretation of thread priorities when deciding which thread to schedule. More precisely: a thread that is not blocked waiting for some resource can never be preempted by a lower priority thread. This is unlike schedulers found in desktop computers (for instance, vanilla POSIX threads [34] without the real-time extension) where even lower priority threads always get to run, preempting

low priority	high priority
lock(m);	$X \leftarrow$ **islocked**(m);
$Y \leftarrow 1$;	**if** $X = 0$ **then**
$Z \leftarrow 1$;	$Z \leftarrow 2$;
$T \leftarrow Y - Z$;	$Y \leftarrow 2$;
unlock(m)	**yield**

Figure 14: Using priorities to ensure mutual exclusion.

higher priority ones if necessary. Moreover, we consider in this section that only a single thread can execute at a given time — this was not required in Sec. 3. This is the case, for instance, when all the threads share a single processor. In this model, the unblocked thread with the highest priority is always the only one to run. All threads start unblocked and may block, either by locking a mutex that is already locked by another thread, or by yielding voluntarily, which allows lower priority threads to run. Yielding denotes blocking for a non-deterministic amount of time, which is useful to model timers of arbitrary duration and waiting for external resources. A lower priority thread can be preempted when unlocking a mutex if a higher priority thread is waiting for this mutex. It can also be preempted at any point by a yielding higher priority thread that wakes up non-deterministically. Thus, a preempted thread can be made to wait at an arbitrary program point, and not necessarily at a synchronization statement. The scheduling is dynamic and the number of possible thread interleavings authorized by the scheduler remains very large, despite being controlled by strict priorities.

This scheduling model is precise enough to take into account fine mutual exclusion properties that would not hold if we considered arbitrary preemption or true parallel executions on concurrent processors. For instance, in Fig. 14, the high priority thread avoids a call to **lock** by testing with **islocked** whether the low priority thread acquired the lock and, if not, executes its critical section and modifies Y and Z, confident that the low priority thread cannot execute and enter its critical section before the high priority thread explicitly **yield**s.

4.2. **Concrete Scheduled Interleaving Semantics $\mathbb{P}_{\mathcal{H}}$.** We now refine the various semantics of Sec. 3 to take scheduling into account, starting with the concrete interleaving semantics $\mathbb{P}_*$ of Sec. 3.1. In this case, it is sufficient to redefine the semantics of primitive statements. This new semantics will, in particular, exclude interleavings that do not respect mutual exclusion or priorities, and thus, we observe fewer behaviors. This is materialized by the dotted $\subseteq$ arrow in Fig. 1 between $\mathbb{P}_*$ and the refined semantics $\mathbb{P}_{\mathcal{H}}$ we are about to present.[4]

We define a domain of scheduler states $\mathcal{H}$ as follows:

$$\mathcal{H} \stackrel{\text{def}}{=} (\mathcal{T} \to \{\, \mathit{ready}, \mathit{yield}, \mathit{wait}(m) \mid m \in \mathcal{M} \,\}) \times (\mathcal{T} \to \mathcal{P}(\mathcal{M})) \ . \tag{4.2}$$

A scheduler state $(b, l) \in \mathcal{H}$ is a pair, where the function b associates to each thread whether it is ready (i.e., it is not blocked, and runs if no higher priority thread is also ready), yielding

[4]Note that Fig. 1 states that each concrete semantics without scheduler abstracts the corresponding concrete semantics with scheduler, but states nothing about abstract semantics. Abstract semantics are generally incomparable due to the use of non-monotonic abstractions and widenings.

(i.e., it is blocked at a **yield** statement), or waiting for some mutex m (i.e., it is blocked at a **lock**(m) statement). The function l associates to each thread the set of all the mutexes it holds. A program state is now a pair $((b,l),\rho)$ composed of a scheduler state $(b,l)\in\mathcal{H}$ and an environment $\rho\in\mathcal{E}$. The semantic domain $\mathcal{D}\stackrel{\text{def}}{=}\mathcal{P}(\mathcal{E})\times\mathcal{P}(\mathcal{L})$ from (2.1) is thus replaced with $\mathcal{D}_\mathcal{H}$ defined as:

$$\mathcal{D}_\mathcal{H}\stackrel{\text{def}}{=}\mathcal{P}(\mathcal{H}\times\mathcal{E})\times\mathcal{P}(\mathcal{L}) \tag{4.3}$$

with the associated pairwise join $\sqcup_\mathcal{H}$.

The semantics $\mathbb{S}_\mathcal{H}[\![\, s,t\,]\!]$ of a primitive statement s executed by a thread $t\in\mathcal{T}$ is described in Fig. 15. It is decomposed into three steps: *enabled*$_t$, $\mathbb{S}^\dagger_\mathcal{H}[\![\, s,t\,]\!]$, and *sched*, the first and the last steps being independent from the choice of statement s. Firstly, the function *enabled*$_t$ filters states to keep only those where the thread t can actually run, i.e., t is the highest priority thread which is ready. Secondly, the function $\mathbb{S}^\dagger_\mathcal{H}[\![\, s,t\,]\!]$ handles the statement-specific semantics. For **yield**, **lock**, and **unlock** statements, this consists in updating the scheduler part of each state. For **lock** statements, the thread enters a wait state until the mutex is available. Actually acquiring the mutex is performed by the following *sched* step if the mutex is immediately available, and by a later *sched* step following the unlocking of the mutex by its owner thread otherwise. The **islocked** statement updates each environment according to its paired scheduler state. The other primitive statements, assignments and guards, are not related to scheduling; their semantics is defined by applying the regular, mono-threaded semantics $\mathbb{S}[\![\, s\,]\!]$ from Fig. 4 to the environment part, leaving the scheduler state unchanged. Thirdly, the function *sched* updates the scheduler state by waking up yielding threads non-deterministically, and giving any newly available mutex to the highest priority thread waiting for it, if any.

The semantics $\mathbb{\Pi}_\mathcal{H}[\![\, P\,]\!]\in\mathcal{D}_\mathcal{H}\xrightarrow{\sqcup_\mathcal{H}}\mathcal{D}_\mathcal{H}$ of a set $P\subseteq\Pi_*$ of parallel control paths then becomes, similarly to (3.1):

$$\begin{array}{l}\mathbb{\Pi}_\mathcal{H}[\![\, P\,]\!](R,\Omega)\stackrel{\text{def}}{=}\\ \quad\bigsqcup_\mathcal{H}\{\,(\mathbb{S}_\mathcal{H}[\![\, s_n,t_n\,]\!]\circ\cdots\circ\mathbb{S}_\mathcal{H}[\![\, s_1,t_1\,]\!])(R,\Omega)\mid(s_1,t_1)\cdot\ldots\cdot(s_n,t_n)\in P\,\}\end{array} \tag{4.4}$$

and the semantics $\mathbb{P}_\mathcal{H}$ of the program is, similarly to (3.3):

$$\mathbb{P}_\mathcal{H}\stackrel{\text{def}}{=}\Omega,\ \text{where}\ (-,\Omega)=\mathbb{\Pi}_\mathcal{H}[\![\,\pi_*\,]\!](\{h_0\}\times\mathcal{E}_0,\emptyset) \tag{4.5}$$

where π_* is the set of parallel control paths of the program, defined in (3.2), and $h_0\stackrel{\text{def}}{=}(\lambda t:\mathit{ready},\lambda t:\emptyset)$ denotes the initial scheduler state (all the threads are ready and hold no mutex). As in Sec. 3.1, many parallel control paths in π_* are unfeasible, i.e., return an empty set of environments, some of which are now ruled out by the *enabled*$_t$ function because they do not obey the real-time scheduling policy or do not ensure the mutual exclusion enforced by locks. Nevertheless, errors from a feasible prefix of an unfeasible path are still taken into account. This includes, in particular, the errors that occur before a dead-lock.

4.3. **Scheduled Weakly Consistent Memory Semantics $\mathbb{P}'_\mathcal{H}$.** As was the case for the interleaving semantics without a scheduler (Sec. 3.1), the scheduled interleaving semantics does not take into account the effect of a weakly consistent memory. Recall that a lack of memory consistency can be caused by the underlying hardware memory model of a multi-processor, by compiler optimisations, or by non-atomic primitive statements. While we can disregard the hardware issues when considering mono-processor systems (i.e., everywhere in

$$\underline{\mathbb{S}_{\mathcal{H}}[\![s,t]\!] : \mathcal{D}_{\mathcal{H}} \xrightarrow{\sqcup_{\mathcal{H}}} \mathcal{D}_{\mathcal{H}}}$$

$$\mathbb{S}_{\mathcal{H}}[\![s,t]\!] \stackrel{\text{def}}{=} \mathit{sched} \circ \mathbb{S}^{\dagger}_{\mathcal{H}}[\![s,t]\!] \circ \mathit{enabled}_t$$

where:

$$\mathit{enabled}_t(R,\Omega) \stackrel{\text{def}}{=} (\{\, ((b,l),\rho) \in R \mid b(t) = \mathit{ready} \wedge \forall t' > t : b(t') \neq \mathit{ready} \,\}, \Omega)$$

$$\mathbb{S}^{\dagger}_{\mathcal{H}}[\![\mathbf{yield}, t]\!](R,\Omega) \stackrel{\text{def}}{=} (\{\, ((b[t \mapsto \mathit{yield}], l), \rho) \mid ((b,l),\rho) \in R \,\}, \Omega)$$
$$\mathbb{S}^{\dagger}_{\mathcal{H}}[\![\mathbf{lock}(m), t]\!](R,\Omega) \stackrel{\text{def}}{=} (\{\, ((b[t \mapsto \mathit{wait}(m)], l), \rho) \mid ((b,l),\rho) \in R \,\}, \Omega)$$
$$\mathbb{S}^{\dagger}_{\mathcal{H}}[\![\mathbf{unlock}(m), t]\!](R,\Omega) \stackrel{\text{def}}{=} (\{\, ((b, l[t \mapsto l(t) \setminus \{m\}]), \rho) \mid ((b,l),\rho) \in R \,\}, \Omega)$$
$$\mathbb{S}^{\dagger}_{\mathcal{H}}[\![X \leftarrow \mathbf{islocked}(m), t]\!](R,\Omega) \stackrel{\text{def}}{=}$$
$$\quad (\{\, ((b,l), \rho[X \mapsto 0]) \mid ((b,l),\rho) \in R,\ \forall t' \in \mathcal{T} : m \notin l(t') \,\} \cup$$
$$\quad \{\, ((b,l), \rho[X \mapsto 1]) \mid ((b,l),\rho) \in R,\ \exists t' \in \mathcal{T} : m \in l(t') \,\}, \Omega)$$

for all other primitive statements $s \in \{\, X \leftarrow e, e \bowtie 0? \,\}$:

$$\mathbb{S}^{\dagger}_{\mathcal{H}}[\![s,t]\!](R,\Omega) \stackrel{\text{def}}{=}$$
$$\quad (\{\, ((b,l),\rho') \mid \exists \rho : ((b,l),\rho) \in R,\ (R',-) = \mathbb{S}[\![s]\!](\{\rho\},\Omega),\ \rho' \in R' \,\}, \Omega')$$
$$\quad \text{where } (-,\Omega') = \mathbb{S}[\![s]\!](\{\, \rho \mid (-,\rho) \in R \,\}, \Omega)$$

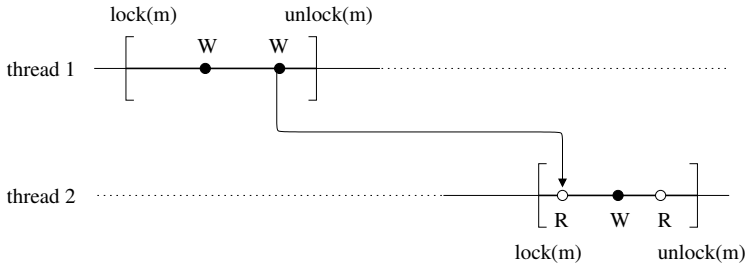
$$\mathit{sched}(R,\Omega) \stackrel{\text{def}}{=} (\{\, ((b',l'),\rho) \mid ((b,l),\rho) \in R \,\}, \Omega)$$
$$\quad \text{s.t. } \forall t :$$
$$\quad\quad \text{if } b(t) = \mathit{wait}(m) \wedge (m \in l(t) \vee (\forall t' : m \notin l(t') \wedge \forall t' > t : b(t') \neq \mathit{wait}(m)))$$
$$\quad\quad \text{then } b'(t) = \mathit{ready} \wedge l'(t) = l(t) \cup \{m\}$$
$$\quad\quad \text{else } l'(t) = l(t) \wedge (b'(t) = b(t) \vee (b'(t) = \mathit{ready} \wedge b(t) = \mathit{yield}))$$

Figure 15: Concrete semantics of primitive statements with a scheduler.

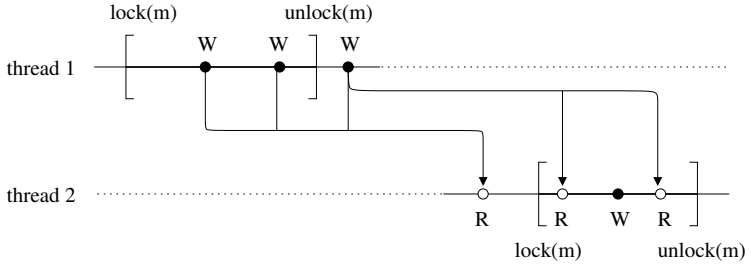
Sec. 4 except Sec. 4.4.5) the other issues remain, and so, we must consider their interaction with the scheduler. Thus, we now briefly present a weakly consistent memory semantics for programs with a scheduler. The interference semantics designed in Secs. 4.4 and 4.5 will be sound with respect to this semantics.

In addition to restricting the interleaving of threads, synchronization primitives also have an effect when considering weakly consistent memory semantics: they enforce some form of sequential consistency at a coarse granularity level. More precisely, the compiler and processor handle synchronization statements specially, introducing the necessary flushes into memory and register reloads, and refraining from optimizing across them.

Recall that the weakly consistent semantics $\mathbb{P}'_*$ of Sec. 3.4 is based on the interleaving semantics $\mathbb{P}_*$ of Sec. 3.1 applied to transformed threads $\pi'(t)$, which are obtained by transforming the paths in $\pi(\mathit{body}_t)$ using elementary path transformations $q \rightsquigarrow q'$ from Def. 3.4. To take synchronization into account, we use the same definition of transformed threads $\pi'(t)$, but restrict it to transformations $q \rightsquigarrow q'$ that do not contain any synchronization primitive. For instance, we forbid the application of sub-expression elimination (Def. 3.4.(8)) on the following path: $\mathbf{lock}(m) \cdot Y \leftarrow e \rightsquigarrow X \leftarrow e \cdot \mathbf{lock}(m) \cdot Y \leftarrow X$. However, if q and q' do not contain any synchronization primitive, and $q \rightsquigarrow q'$, then it is legal to replace q with q' in a path containing synchronization primitives before and after q. For instance, the transformation $\mathbf{lock}(m) \cdot Y \leftarrow e \rightsquigarrow \mathbf{lock}(m) \cdot X \leftarrow e \cdot Y \leftarrow X$ is



(a) Well synchronized communication.



(b) Weakly consistent communications.

Figure 16: Well synchronized versus weakly consistent communications.

acceptable. The scheduled weakly consistent memory semantics is then, based on (4.5):

$$\mathbb{P}'_{\mathcal{H}} \stackrel{\text{def}}{=} \Omega, \text{ where } (-, \Omega) = \mathbb{\Pi}_{\mathcal{H}}[\![\pi'_*]\!](\{h_0\} \times \mathcal{E}_0, \emptyset) \tag{4.6}$$

where π'_* is defined, as before (3.7), as the interleavings of control paths from all $\pi'(t)$, $t \in \mathcal{T}$.

4.4. **Concrete Scheduled Interference Semantics $\mathbb{P}_{\mathcal{C}}$.** We now provide a structured version $\mathbb{P}_{\mathcal{C}}$ of the scheduled interleaving semantics $\mathbb{P}_{\mathcal{H}}$. Similarly to the interference abstraction $\mathbb{P}_{\mathcal{I}}$ from Sec. 3.2 of the non-scheduled interleaving semantics $\mathbb{P}_*$, it is based on a notion of interference, it is sound with respect to both the interleaving semantics $\mathbb{P}_{\mathcal{H}}$ and its weakly consistent version $\mathbb{P}'_{\mathcal{H}}$, but it is not complete with respect to either of them. The main changes with respect to the interference abstraction $\mathbb{P}_{\mathcal{I}}$ are: a notion of scheduler configuration (recording some information about the state of mutexes), a partitioning of interferences and environments with respect to configurations, and a distinction between well synchronized thread communications and data-races. As our semantics is rather complex, we first present it graphically on examples before describing it in formal terms.

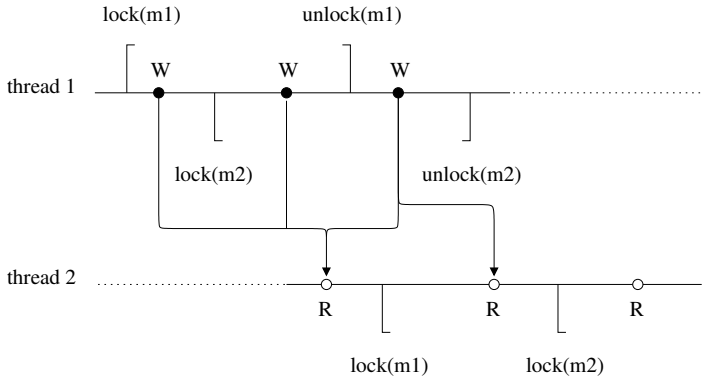
4.4.1. *Interferences.* In the non-scheduled semantics $\mathbb{P}_{\mathcal{I}}$ (Sec. 3.2), any interference (t, X, v), i.e., any write by a thread t of a value v into a variable X, could influence any read from the same variable X in another thread $t' \neq t$. While this is also a sound abstraction of the semantics with a scheduler, the precision can be improved by refining our notion of interference and exploiting mutual exclusion properties enforced by the scheduler.

Good programming practice dictates that all read and write accesses to a given shared variable should be protected by a common mutex. This is exemplified in Fig. 16.(a) where W and R denote respectively a write to and a read from a variable X, and all reads and

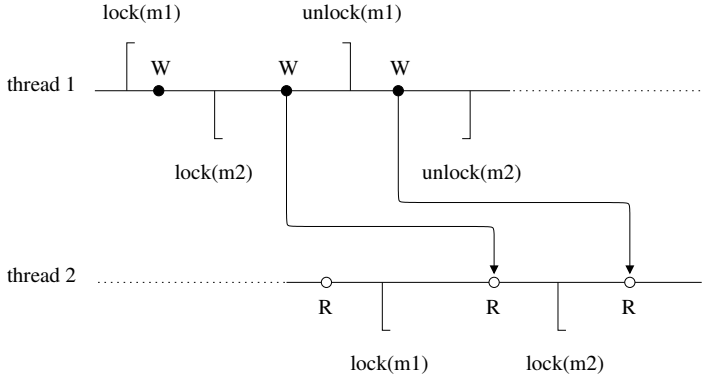
writes are protected by a mutex m. In this example, thread 1 writes twice to X while holding m. Thus, when thread 2 locks m and reads X, it can see the second value written by thread 1, but never the first one, which is necessarily overwritten before thread 2 acquires m. Likewise, after thread 2 locks m and overwrites X, while it still holds m it can only read back the value it has written and not any value written by thread 1. Thus, a single interference from thread 1 can effect thread 2, and at only one read position; we call this read / write pair a "well synchronized communication." Well synchronized communications are flow-sensitive (the order of writes and reads matters), and so, differ significantly from the interferences of Sec. 3.2. In practice, we model such communications by recording at the point **unlock**(m) in thread 1 the current value of all the variables that are modified while m is locked, and import these values in the environments at the point **lock**(m) in thread 2.

Accesses are not always protected by mutexes, though. Consider, for instance, the example in Fig. 16.(b), where X may additionally be modified by thread 1 and read by thread 2 outside the critical sections defined by mutex m. In addition to the well synchronized communication of Fig. 16.(a), which is omitted for clarity in Fig. 16.(b), we consider that a write from thread 1 effects a read from thread 2 if either operation is performed while m is not locked. These read / write pairs correspond to data-races, and neither the compiler nor the hardware is required to enforce memory consistency. We call these pairs "weakly consistent communications." In practice, these are handled in a way similar to the interferences in Sec. 3.2: the values thread 1 can write into X are remembered in a flow-insensitive interference set, and the semantics of expressions is modified so that, when reading X in thread 2, either the thread's value for X or a value from the interference set is used. We also remember the set of mutexes that threads hold during each read and each write, so that we can discard communications that cannot occur due to mutual exclusion. For instance, in Fig. 16.(b), there is no communication of any kind between the first write in thread 1 and the second read in thread 2. The example also shows that well synchronized and weakly consistent communications can mix freely: there is no weakly consistent communication between the second write in thread 1 and the second read in thread 2 due to mutual exclusion (both threads hold the mutex m); however, there is a well synchronized communication — shown in Fig. 16.(a).

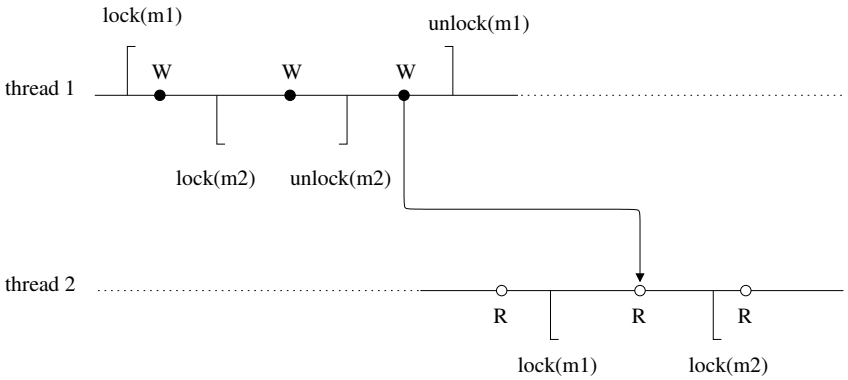
Figure 17 illustrates the communications in the case of several mutexes: $m1$ and $m2$. In Fig. 17.(a), weakly consistent communications only occur between write / read pairs when the involved threads have not locked a common mutex. For instance, the first write by thread 1 is tagged with the set of locked mutexes $\{m1\}$, and so, can only influence the first read by thread 2 (tagged with $\emptyset$) and not the following two (tagged respectively with $\{m1\}$ and $\{m1, m2\}$). Likewise, the second write, tagged with $\{m1, m2\}$, only influences the first read. However, the third write, tagged with only $\{m2\}$, influences the two first reads (thread 2 does not hold the mutex $m2$ there). In Fig. 17.(b), well synchronized communications import, as before, at a lock of mutex $m1$ (resp. $m2$) in thread 2, the last value written by thread 1 before unlocking the same mutex $m1$ (resp. $m2$). The well synchronized communication in Fig. 17.(c) is more interesting. In that case, thread 1 unlocks $m2$ before $m1$, instead of after. As expected, when thread 2 locks $m1$, it imports the last (third) value written by thread 1, just before unlocking $m1$. We note, however, that the second write in thread 1 does not influence thread 2 while thread 2 holds mutex $m1$, as the value is always over-written by thread 1 before unlocking $m1$. We model this by importing, when locking a mutex m in thread 2, only the values written by thread 1 while it does not



(a) Weakly consistent communications.



(b) Well synchronized communications.



(c) Well synchronized communications.

Figure 17: Well synchronized and weakly consistent communications with two locks.

hold a common mutex (in addition to m) with thread 2. Thus, when locking $m2$ while still holding the mutex $m1$, thread 2 does not import the second value written by thread 1 because thread 1 also holds $m1$ during this write.

4.4.2. *Interference partitioning.* To differentiate between well synchronized and weakly consistent communications, and to avoid considering communications between parts of threads that are in mutual exclusion, we partition interferences with respect to a thread-local view of scheduler configurations. The (finite) set $\mathcal{C}$ of configurations is defined as:

$$\mathcal{C} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{M}) \times \mathcal{P}(\mathcal{M}) \times \{\, \mathit{weak}, \mathit{sync}(m) \mid m \in \mathcal{M} \,\} \quad . \tag{4.7}$$

In a configuration $(l, u, s) \in \mathcal{C}$, the first component $l \subseteq \mathcal{M}$ denotes the exact set of mutexes locked by the thread creating the interference, which is useful to decide which reads will be affected by the interference. The second component $u \subseteq \mathcal{M}$ denotes a set of mutexes that are known to be locked by no thread (either current or not). This information is inferred by the semantics of **islocked** statements and can be exploited to detect extra mutual exclusion properties that further limit the set of reads affected by an interference (as in the example in Fig. 14). The last component, s, allows distinguishing between weakly consistent and well synchronized communications: *weak* denotes an interference that generates weakly consistent communications, while $\mathit{sync}(m)$ denotes an interference that generates well synchronized communications for critical sections protected by the mutex m. These two kinds of interferences are naturally called, respectively, weakly consistent and well synchronized interferences. The partitioned domain of interferences is then:

$$\mathcal{I} \stackrel{\text{def}}{=} \mathcal{T} \times \mathcal{C} \times \mathcal{V} \times \mathbb{R} \tag{4.8}$$

which enriches the definition of $\mathcal{I}$ from Sec. 3.2 with a scheduler configuration in $\mathcal{C}$. The interference $(t, c, X, v) \in \mathcal{I}$ indicates that the thread $t \in \mathcal{T}$ can write the value $v \in \mathbb{R}$ into the variable $X \in \mathcal{V}$ and the scheduler is in the configuration $c \in \mathcal{C}$ at the time of the write.

4.4.3. *Environment partitioning.* When computing program states in our semantics, environments are also partitioned with respect to scheduler configurations in order to track some information on the current state of mutexes. Thus, our program states associate an environment $\rho \in \mathcal{E}$ and a configuration in $(l, u, s) \in \mathcal{C}$, where the configuration (l, u, s) indicates the set of mutexes l held by the thread in that state, as well as the set of mutexes u that are known to be held by no thread; the s component is not used and always set by convention to *weak*. The semantic domain is now:

$$\mathcal{D}_{\mathcal{C}} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{C} \times \mathcal{E}) \times \mathcal{P}(\mathcal{L}) \times \mathcal{P}(\mathcal{I}) \tag{4.9}$$

partially ordered by pointwise set inclusion. We denote by $\sqcup_{\mathcal{C}}$ the associated pointwise join. While regular statements (such as assignments and tests) update the environment part of each state, synchronization primitives update the scheduler part of the state.

The use of pairs of environments and scheduler configurations allows representing relationships between the value of a variable and the state of a mutex, which is important for the precise modeling of the **islocked** primitive in code similar to that of Fig. 14. For instance, after the statement $X \leftarrow \textbf{islocked}(m)$, all states $((l, u, s), \rho)$ satisfy $\rho(X) = 0 \Longrightarrow m \in u$. Thus, when the high thread enters the "then" branch of the subsequent $X = 0$ test, we know that m is not locked by any thread and we can disregard the interferences generated by the low thread while holding m.

$$\underline{\mathbb{E}_{\mathcal{C}}[\![\, e \,]\!] : (\mathcal{T} \times \mathcal{C} \times \mathcal{E} \times \mathcal{P}(\mathcal{I})) \to (\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\mathcal{L}))}$$

$$\mathbb{E}_{\mathcal{C}}[\![\, X \,]\!](t, c, \rho, I) \stackrel{\text{def}}{=} (\{\, \rho(X) \,\} \cup \{\, v \mid \exists (t', c', X, v) \in I : t \neq t' \wedge \mathit{intf}(c, c') \,\}, \emptyset)$$

$$\mathbb{E}_{\mathcal{C}}[\![\, [c_1, c_2] \,]\!](t, c, \rho, I) \stackrel{\text{def}}{=} (\{\, c \in \mathbb{R} \mid c_1 \leq c \leq c_2 \,\}, \emptyset)$$

$$\mathbb{E}_{\mathcal{C}}[\![\, -_\ell\ e \,]\!](t, c, \rho, I) \stackrel{\text{def}}{=} \text{let } (V, \Omega) = \mathbb{E}_{\mathcal{C}}[\![\, e \,]\!](t, c, \rho, I) \text{ in } (\{\, -x \mid x \in V \,\}, \Omega)$$

$$\begin{array}{l}\mathbb{E}_{\mathcal{C}}[\![\, e_1 \diamond_\ell e_2 \,]\!](t, c, \rho, I) \stackrel{\text{def}}{=} \\ \quad \text{let } (V_1, \Omega_1) = \mathbb{E}_{\mathcal{C}}[\![\, e_1 \,]\!](t, c, \rho, I) \text{ in} \\ \quad \text{let } (V_2, \Omega_2) = \mathbb{E}_{\mathcal{C}}[\![\, e_2 \,]\!](t, c, \rho, I) \text{ in} \\ \quad (\{\, x_1 \diamond x_2 \mid x_1 \in V_1,\ x_2 \in V_2,\ \diamond \neq / \vee x_2 \neq 0 \,\}, \\ \quad\ \Omega_1 \cup \Omega_2 \cup \{\, \ell \mid \diamond = / \wedge 0 \in V_2 \,\}) \\ \text{where } \diamond \in \{\, +, -, \times, / \,\}\end{array}$$

where:

$$\mathit{intf}((l, u, s), (l', u', s')) \stackrel{\text{def}}{\iff} l \cap l' = u \cap l' = u' \cap l = \emptyset \wedge s = s' = \mathit{weak}$$

Figure 18: Concrete scheduled semantics of expressions with interference.

4.4.4. *Semantics.* We now describe in details the semantics of expressions and statements. It is presented fully formally in Figs. 18 and 19.

The semantics $\mathbb{E}_{\mathcal{C}}[\![\, e \,]\!](t, c, \rho, I)$ of an expression e is presented in Fig. 18. It is similar to the non-scheduled semantics $\mathbb{E}_{\mathcal{I}}[\![\, e \,]\!](t, \rho, I)$ of Fig. 7, except that it has an extra argument: the current configuration $c \in \mathcal{C}$ (4.7) of the thread evaluating the expression. The other arguments are: the thread $t \in \mathcal{T}$ evaluating the expression, the environment $\rho \in \mathcal{E}$ in which it is evaluated, and a set $I \subseteq \mathcal{I}$ (4.8) of interferences from the whole program. Interferences are applied when reading a variable $\mathbb{E}_{\mathcal{C}}[\![\, X \,]\!]$. Only weakly consistent interferences are handled in expressions — well synchronized interferences are handled in the semantics of synchronization primitives, presented below. Moreover, we consider only interferences with configurations that are not in mutual exclusion with the current configuration c. Mutual exclusion is enforced by the predicate intf, which states that, in two scheduler configurations (l, u, weak) and (l', u', weak) for distinct threads, no mutex can be locked by both threads $(l \cap l' = \emptyset)$, and no thread can lock a mutex which is assumed to be free by the other one $(l \cap u' = l' \cap u = \emptyset)$. The semantics of other expression constructs remains the same, passing recursively the arguments t, c, and I unused and unchanged.

We now turn to the semantics $\mathbb{S}_{\mathcal{C}}[\![\, s, t \,]\!](R, \Omega, I)$ of a statement s executed by a thread t, which is defined in Fig. 19. It takes as first argument a set R of states which are now pairs consisting of an environment $\rho \in \mathcal{E}$ and a scheduler configuration $c \in \mathcal{C}$, i.e., $R \subseteq \mathcal{C} \times \mathcal{E}$. The other arguments are, as in the non-scheduled semantics of Fig. 8, a set of run-time errors $\Omega \subseteq \mathcal{L}$ to enrich, and a set of interferences $I \subseteq \mathcal{I}$ to use and enrich. The semantics of assignments and tests in Fig. 19 is similar to the non-scheduled case (Fig. 8). The scheduler configuration associated with each input environment is simply passed as argument to the expression semantics $\mathbb{E}_{\mathcal{C}}$ in order to select precisely which weakly relational interferences to apply (through intf), but it is otherwise left unmodified in the output. Additionally, assignments $X \leftarrow e$ generate weakly consistent interferences, which store in I the current thread t and the scheduler configuration c of its state, in addition to the modified variable X and its new value.

$\underline{\mathbb{S}_{\mathcal{C}}[\![s,t]\!] : \mathcal{D}_{\mathcal{C}} \xrightarrow{\sqcup_{\mathcal{C}}} \mathcal{D}_{\mathcal{C}}}$

$$\mathbb{S}_{\mathcal{C}}[\![X \leftarrow e,\, t]\!](R,\Omega,I) \stackrel{\text{def}}{=} (\emptyset,\Omega,I) \sqcup_{\mathcal{C}} \bigsqcup_{(c,\rho)\in R}^{\mathcal{C}} \text{let } (V,\Omega') = \mathbb{E}_{\mathcal{C}}[\![e]\!](t,c,\rho,I) \text{ in } (\{\,(c,\rho[X\mapsto v]) \mid v\in V\,\},\Omega',\{\,(t,c,X,v)\mid v\in V\,\})$$

$$\mathbb{S}_{\mathcal{C}}[\![e \bowtie 0?,\, t]\!](R,\Omega,I) \stackrel{\text{def}}{=} (\emptyset,\Omega,I) \sqcup_{\mathcal{C}} \bigsqcup_{(c,\rho)\in R}^{\mathcal{C}} \text{let } (V,\Omega') = \mathbb{E}_{\mathcal{C}}[\![e]\!](t,c,\rho,I) \text{ in } (\{\,(c,\rho) \mid \exists v\in V : v\bowtie 0\,\},\Omega',\emptyset)$$

$$\mathbb{S}_{\mathcal{C}}[\![\textbf{if } e\bowtie 0 \textbf{ then } s,\, t]\!](R,\Omega,I) \stackrel{\text{def}}{=} (\mathbb{S}_{\mathcal{C}}[\![s,t]\!]\circ\mathbb{S}_{\mathcal{C}}[\![e\bowtie 0?,\, t]\!])(R,\Omega,I) \sqcup_{\mathcal{C}} \mathbb{S}_{\mathcal{C}}[\![e\not\bowtie 0?,\, t]\!](R,\Omega,I)$$

$$\mathbb{S}_{\mathcal{C}}[\![\textbf{while } e\bowtie 0 \textbf{ do } s,\, t]\!](R,\Omega,I) \stackrel{\text{def}}{=} \mathbb{S}_{\mathcal{C}}[\![e\not\bowtie 0?,\, t]\!](\mathit{lfp}\,\lambda X : (R,\Omega,I) \sqcup_{\mathcal{C}} (\mathbb{S}_{\mathcal{C}}[\![s,t]\!]\circ\mathbb{S}_{\mathcal{C}}[\![e\bowtie 0?,\, t]\!])X)$$

$$\mathbb{S}_{\mathcal{C}}[\![s_1;\, s_2,\, t]\!](R,\Omega,I) \stackrel{\text{def}}{=} (\mathbb{S}_{\mathcal{C}}[\![s_2,t]\!]\circ\mathbb{S}_{\mathcal{C}}[\![s_1,t]\!])(R,\Omega,I)$$

$$\mathbb{S}_{\mathcal{C}}[\![\textbf{lock}(m),\, t]\!](R,\Omega,I) \stackrel{\text{def}}{=} (\{\,((l\cup\{m\},\emptyset,s),\rho') \mid ((l,-,s),\rho)\in R,\, \rho'\in \mathit{in}(t,l,\emptyset,m,\rho,I)\,\}, \Omega,\, I\cup\bigcup\{\,\mathit{out}(t,l,\emptyset,m',\rho,I) \mid \exists u : ((l,u,-),\rho)\in R\wedge m'\in u\,\})$$

$$\mathbb{S}_{\mathcal{C}}[\![\textbf{unlock}(m),\, t]\!](R,\Omega,I) \stackrel{\text{def}}{=} (\{\,((l\setminus\{m\},u,s),\rho) \mid ((l,u,s),\rho)\in R\,\}, \Omega,\, I\cup\bigcup\{\,\mathit{out}(t,l\setminus\{m\},u,m,\rho,I) \mid ((l,u,-),\rho)\in R\,\})$$

$$\mathbb{S}_{\mathcal{C}}[\![\textbf{yield},\, t]\!](R,\Omega,I) \stackrel{\text{def}}{=} (\{\,((l,\emptyset,s),\rho) \mid ((l,-,s),\rho)\in R\,\}, \Omega,\, I\cup\bigcup\{\,\mathit{out}(t,l,\emptyset,m',\rho,I) \mid \exists u : ((l,u,-),\rho)\in R\wedge m'\in u\,\})$$

$$\mathbb{S}_{\mathcal{C}}[\![X \leftarrow \textbf{islocked}(m),\, t]\!](R,\Omega,I) \stackrel{\text{def}}{=}$$

if no thread $t' > t$ locks m, then:

$$(\{\,((l,u\cup\{m\},s),\rho'[X\mapsto 0]) \mid ((l,u,s),\rho)\in R,\, \rho'\in \mathit{in}(t,l,u,m,\rho,I)\,\}\cup \{\,((l,u\setminus\{m\},s),\rho[X\mapsto 1]) \mid ((l,u,s),\rho)\in R\,\}, \Omega,\, I\cup\{\,(t,c,X,v)\mid v\in\{0,1\},\,(c,-)\in R\,\})$$

otherwise:

$$\mathbb{S}_{\mathcal{C}}[\![X\leftarrow[0,1],\, t]\!](R,\Omega,I)$$

where:

$$\mathit{in}(t,l,u,m,\rho,I) \stackrel{\text{def}}{=} \{\,\rho' \mid \forall X\in\mathcal{V} : \rho'(X)=\rho(X)\vee(\exists t',l',u' : (t',(l',u',\mathit{sync}(m)),X,\rho'(X))\in I \wedge t\neq t'\wedge l\cap l' = l\cap u' = l'\cap u = \emptyset)\,\}$$

$$\mathit{out}(t,l,u,m,\rho,I) \stackrel{\text{def}}{=} \{\,(t,(l,u,\mathit{sync}(m)),X,\rho(X)) \mid \exists l' : (t,(l',-,\mathit{weak}),X,-)\in I\wedge m\in l'\,\}$$

Figure 19: Concrete scheduled semantics of statements with interference.

The semantics of non-primitive statements remains the same as in previous semantics by structural induction on the syntax of statements (e.g., Fig. 8).

The main point of note is thus the semantics of synchronization primitives. It updates the scheduler configuration and takes care of well synchronized interferences.

Let us explain first how the scheduler part $(l, u, s) \in \mathcal{C}$ of a state $((l, u, s), \rho) \in R$ is updated. Firstly, the set l of mutexes held by the current thread is updated by the primitives **lock**(m) and **unlock**(m) by respectively adding m to and removing m from l. Secondly, the set of mutexes u that are known to be free in the system is updated by $X \leftarrow$ **islocked**(m). Generally, no information on the state of the mutex is known *a priori*. Each input state thus spawns two output states: one where m is free ($m \in u$), and one where m is not free ($m \notin u$). In the first state, X is set to 0 while, in the second state, it is set to 1. As a consequence, although the primitive cannot actually infer whether the mutex is free or not, it nevertheless keeps the relationship between the value of X and the fact that m is free. Inferring this relation is sufficient to analyze precisely the code in Fig. 14. It is important to note that the information in $m \in u$ is transient as, when a context switch occurs, another thread t' can run and lock m, thus invaliding the assumption by thread t that no thread has locked m. We distinguish two scenarios, depending on whether t' has higher priority than t or not. When $t' < t$, the thread t' has lower priority and cannot preempt t at an arbitrary point due to the real-time nature of the scheduler. Instead, t' must wait until t performs a blocking operation (i.e., calls a **lock** or **yield** primitive) to get the opportunity to lock m. This case is handled by having all our blocking primitives reset the u component to $\emptyset$. When $t' > t$, the thread t' can preempt t at arbitrary points, including just after the **islocked** primitive, and so, we can never safely assume that $m \in u$. If this scenario is possible, $X \leftarrow$ **islocked**(m) is modeled as $X \leftarrow [0, 1]$, without updating u. To decide which transfer function to use for **islocked**, we need to know the set of all mutexes than can be locked by each thread. It is quite easy to enrich our semantics to track this information but, as it is cumbersome, we did not include this in Fig. 19 — one way is to add a new component $M : \mathcal{T} \to \mathcal{P}(\mathcal{M})$ to the domain $\mathcal{I}$ of interferences, in which we remember the set of arguments m of each **lock**(m) encountered by each thread; then, we check that $\nexists t' > t : m \in M(t')$ before applying the precise transfer function for $X \leftarrow$ **islocked**(m) in thread t.

We now discuss how synchronization primitives handle well synchronized interferences. We use two auxiliary functions, $\mathit{in}(t, l, u, m, \rho, I)$ and $\mathit{out}(t, l, u, m, \rho, I)$, that model respectively entering and exiting a critical section protected by a mutex $m \in \mathcal{M}$ in a thread $t \in \mathcal{T}$. The arguments $l, u \subseteq \mathcal{M}$ reflect the scheduler configuration when the primitive is called, i.e., they are respectively the set of mutexes held by thread t and those assumed to be free in the system. The function $\mathit{out}(t, l, u, m, \rho, I)$ *collects* a set of well synchronized interferences from an environment $\rho \in \mathcal{E}$. These are constructed from the current value $\rho(X)$ of the variables X that have been modified while the mutex m was held. Such information can be tracked precisely in the semantics by adding another component in $\mathcal{C} \to \mathcal{P}(\mathcal{V})$ to our program states R but, for the sake of simplicity, the semantics we present simply extracts this information from the interferences in I: we consider all the variables that have some weakly consistent interference by thread t in a configuration where it holds m ($m \in l$). This may actually over-approximate the set of variables we seek as it includes variables that have been modified in previous critical sections protected by the same mutex m, but not in the current critical section.[5] Given a variable X, the interference we store is then $(t, (l, u, \mathit{sync}(m)), X, \rho(X))$. The function $\mathit{in}(t, l, u, m, \rho, I)$ *applies* well synchronized interferences from I to an environment ρ: it returns all the environments ρ' that can be obtained

[5]Our prototype performs the same over-approximation for the sake of keeping the analysis simple, and we did not find any practical occurrence where this resulted in a loss of precision. We explain this by remaking that critical sections delimited by the same mutex tend to protect the same set of modified variables.

from ρ by setting one or several variables to their interference value. It only considers well synchronized interferences with configuration $\mathit{sync}(m)$ and from threads $t' \neq t$. Moreover, it uses a test similar to that of intf to avoid applying interferences that cannot occur due to mutual exclusion, by comparing the current state of mutexes (l and u) to their state when the interference was stored.

The function pair in / out is actually used to implement *two* kinds of critical sections. A first kind stems from the use of **lock**(m) and **unlock**(m) statements, which naturally delimit a critical section protected by m. Additionally, whenever a mutex m is added to the u scheduler component by a primitive $X \leftarrow$ **islocked**(m), we also enter a critical section protected by m. Thus, in is called for mutex m, and intf ensures that weakly consistent interferences where m is locked are no longer applied. Such critical sections end when m leaves u, that is, whenever the thread executes a blocking primitive: **lock** or **yield**. These primitives call out for every mutex currently in u, and reset u to $\emptyset$ in the program state.

Finally, we turn to the semantics $\mathbb{P}_{\mathcal{C}}$ of a program, which has the same fixpoint form as $\mathbb{P}_{\mathcal{I}}$ (3.4):

$$\begin{aligned}\mathbb{P}_{\mathcal{C}} &\overset{\text{def}}{=} \Omega, \text{ where } (\Omega, -) \overset{\text{def}}{=} \mathit{lfp}\, \lambda(\Omega, I) : \\ &\quad \bigsqcup_{t \in \mathcal{T}} \text{let } (-, \Omega', I') = \mathbb{S}_{\mathcal{C}}[\![\, \mathit{body}_t,\, t \,]\!]\, (\{c_0\} \times \mathcal{E}_0, \Omega, I) \text{ in } (\Omega', I')\end{aligned} \tag{4.10}$$

where the initial configuration is $c_0 \overset{\text{def}}{=} (\emptyset, \emptyset, \mathit{weak}) \in \mathcal{C}$. This semantics is sound with respect to that of Secs. 4.2–4.3:

Theorem 4.1. *$\mathbb{P}_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$ and $\mathbb{P}'_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$.*

Proof. In Appendix A.8. □

4.4.5. *Multiprocessors and non-real-time systems.* The only part of our semantics that exploits the fact that only one thread can execute at a given time is the semantics of $X \leftarrow$ **islocked**(m). It assumes that, after the current thread has performed the test, the state of the mutex m cannot change until the current thread calls a blocking primitive (**lock** or **yield**) — unless some higher priority thread can also lock the mutex m. Thus, in order to obtain a semantics that is also sound for truly parallel or non-real-time systems, it is sufficient to interpret all statements $X \leftarrow$ **islocked**(m) as $X \leftarrow [0, 1]$.

While more general, this semantics is less precise when analyzing a system that is known to be mono-processor and real-time. For instance, this semantics cannot prove that the two threads in Fig. 14 are in mutual exclusion and that, as a result, $T = 0$ at the end of the program. It finds instead $T \in \{-1, 0, 1\}$, which is less precise. As our target application (Sec. 5) is mono-processor and real-time, we will not discuss this more general but less precise semantics further.

4.4.6. *Detecting data-races.* In our semantics, data-races silently cause weakly consistent interferences but are otherwise not reported. It is easy to modify the semantics to output them. Write / write data-races can be directly extracted from the computed set of interferences I gathered by the least fixpoint in (4.10) as follows:

$$\{\, (t, t', X) \in \mathcal{T} \times \mathcal{T} \times \mathcal{V} \mid \exists c, c' : (t, c, X, -) \in I \wedge (t', c', X, -) \in I \wedge t \neq t' \wedge \mathit{intf}(c, c') \,\}$$

is a set where each element (t, t', X) indicates that threads t and t' may both write into X at the same time. Read / write data-races cannot be similarly extracted from I as the

$$\underline{\mathbb{E}_{\mathcal{CR}}[\![\, e \,]\!] : (\mathcal{T} \times \mathcal{C} \times \mathcal{P}(\mathcal{I})) \to \mathcal{P}(\mathcal{T} \times \mathcal{T} \times \mathcal{V})}$$

$$\mathbb{E}_{\mathcal{CR}}[\![\, X \,]\!](t, c, I) \stackrel{\text{def}}{=} \{\, (t, t', X) \mid \exists c' : (t', c', X, -) \in I \wedge t \neq t' \wedge \mathit{intf}(c, c') \,\}$$

$$\mathbb{E}_{\mathcal{CR}}[\![\, [c_1, c_2] \,]\!](t, c, I) \stackrel{\text{def}}{=} \emptyset$$

$$\mathbb{E}_{\mathcal{CR}}[\![\, -_\ell \; e \,]\!](t, c, I) \stackrel{\text{def}}{=} \mathbb{E}_{\mathcal{CR}}[\![\, e \,]\!]\,(t, c, I)$$

$$\mathbb{E}_{\mathcal{CR}}[\![\, e_1 \diamond_\ell e_2 \,]\!](t, c, I) \stackrel{\text{def}}{=} \mathbb{E}_{\mathcal{CR}}[\![\, e_1 \,]\!]\,(t, c, I) \cup \mathbb{E}_{\mathcal{CR}}[\![\, e_2 \,]\!](t, c, I)$$
where $\diamond \in \{\, +, -, \times, / \,\}$

Figure 20: Read / write data-race detection.

$$\mathcal{E}_0 : X = Y = 5$$

thread t_1	thread t_2
while $0 = 0$ **do**	**while** $0 = 0$ **do**
lock(m);	**lock**(m);
if $X > 0$ **then**	**if** $X < 10$ **then**
$X \leftarrow X - 1$;	$X \leftarrow X + 1$;
$Y \leftarrow Y - 1$;	$Y \leftarrow Y + 1$;
unlock(m)	**unlock**(m)

Figure 21: Imprecisely analyzed program due to the lack of relational lock invariant.

set of interferences does not remember which variables are read from, only which ones are written to. A simple solution is to instrument the semantics of expressions so that, during expression evaluation, it gathers the set of read variables that are affected by an interference. This is performed, for instance, by $\mathbb{E}_{\mathcal{CR}}$ presented in Fig. 20. This function has the same arguments as $\mathbb{E}_{\mathcal{C}}$, except that no environment ρ is needed, and it outputs a set of data-races (t, t', X) instead of environments and errors.

4.4.7. *Precision.* The interference abstraction we use in $\mathbb{P}_{\mathcal{C}}$ is sound be not complete with respect to the interleaving-based semantics $\mathbb{P}_{\mathcal{H}}$. In addition to the incompleteness already discussed in Sec. 3.2, some loss of precision comes from the handling of well synchronized accesses. A main limitation is that such accesses are handled in a non-relational way, hence $\mathbb{P}_{\mathcal{C}}$ cannot represent relations enforced at the boundary of critical sections but broken within, while $\mathbb{P}_{\mathcal{H}}$ can. For instance, in Fig. 14, we cannot prove that $Y = Z$ holds outside critical sections, but only that $Y, Z \in \{1, 2\}$. This shows in particular that even programs without data-races have behaviors in $\mathbb{P}_{\mathcal{C}}$ outside the sequentially consistent ones. However, we can prove that the assignment into T is free from interference, and so, that $T = 0$. By contrast, the interference semantics $\mathbb{P}_{\mathcal{I}}$ of Sec. 3.2 ignores synchronization and would output $T \in \{\, -1, 0, 1 \,\}$, which is less precise.

Figure 21 presents another example where the lack of relational interference results in a loss of precision. This example implements an abstract producer / consumer system, where a variable X counts the number of resources, thread t_1 consumes resources ($X \leftarrow X - 1$) if available ($X > 0$), and thread t_2 generates resources ($X \leftarrow X + 1$) if there is still room for resources ($X < 10$). Our interference semantics can prove that X is always bounded in $[0, 10]$. However, it cannot provide an upper bound on the variable Y. Actually, Y is also

$$\mathcal{E}_0 : X = Y = 0$$

high thread	low thread
$X \leftarrow 1;$	$B \leftarrow 1/X;$
$A \leftarrow 1/(Y-1);$	$Y \leftarrow 1$
yield	

Figure 22: Imprecisely analyzed program due to the lack of inter-thread flow-sensitivity.

$$\begin{aligned}
&\mathcal{I}^\sharp \overset{\text{def}}{=} (\mathcal{T} \times \mathcal{C} \times \mathcal{V}) \to \mathcal{N}^\sharp\\
&\gamma_\mathcal{I} : \mathcal{I}^\sharp \to \mathcal{P}(\mathcal{I})\\
&\qquad \text{s.t. } \gamma_\mathcal{I}(I^\sharp) \overset{\text{def}}{=} \{\, (t,c,X,v) \mid t \in \mathcal{T},\, c \in \mathcal{C},\, X \in \mathcal{V},\, v \in \gamma_\mathcal{N}(I^\sharp(t,c,X)) \,\}\\
&\bot^\sharp_\mathcal{I} \overset{\text{def}}{=} \lambda(t,c,X) : \bot^\sharp_\mathcal{N}\\
&I^\sharp_1 \cup^\sharp_\mathcal{I} I^\sharp_2 \overset{\text{def}}{=} \lambda(t,c,X) : I^\sharp_1(t,c,X) \cup^\sharp_\mathcal{N} I^\sharp_2(t,c,X)\\
&I^\sharp_1 \triangledown_\mathcal{I} I^\sharp_2 \overset{\text{def}}{=} \lambda(t,c,X) : I^\sharp_1(t,c,X) \triangledown_\mathcal{N} I^\sharp_2(t,c,X)
\end{aligned}$$

Figure 23: Abstract domain of scheduled interferences $\mathcal{I}^\sharp$, derived from $\mathcal{N}^\sharp$.

bounded by $[0, 10]$ as it mirrors X. Proving this would require inferring a relational lock invariant: $X = Y$.

Finally, Fig. 22 presents an example where the lack of inter-thread flow sensitivity results in a loss of precision. In this example, the high priority thread always executes first, until it reaches **yield**, at which point it allows the lower priority thread to execute. To prove that the expression $1/(Y-1)$ does not perform an error, it is necessary to prove that it is executed before the low thread stores 1 into Y. Likewise, to prove that the expression $1/X$ does not perform an error in the low thread, it is necessary to prove that it is executed after the high thread stores 1 into X. With respect to flow sensitivity, our semantics is only able to express that an event is performed before another one within the same thread (intra-thread flow sensitivity) and that a thread communication between a pair of locations cannot occur (mutual exclusion), but it is not able to express that an event in a thread is performed before another one in another thread (inter-thread flow sensitivity).

4.5. **Abstract Scheduled Interference Semantics $\mathbb{P}^\sharp_\mathcal{C}$.** We now abstract the interference semantics with scheduler $\mathbb{P}_\mathcal{C}$ from the preceding section in order to construct an effective static analyzer. We reuse the ideas from the abstraction $\mathbb{P}^\sharp_\mathcal{I}$ of $\mathbb{P}_\mathcal{I}$ in Sec. 3.3. The main difference is that we track precisely scheduler configurations in $\mathcal{C}$ (4.7), and we partition abstract environments and interferences with respect to them.

As in Sec. 3.3, we assume that an abstract domain $\mathcal{E}^\sharp$ of environment sets $\mathcal{P}(\mathcal{E})$ is given (with signature in Fig. 5), as well as an abstract domain $\mathcal{N}^\sharp$ of real sets $\mathcal{P}(\mathbb{R})$ (with signature in Fig. 10). The abstract domain of interferences $\mathcal{I}^\sharp$, abstracting $\mathcal{I}$ (4.8), is obtained by partitioning $\mathcal{N}^\sharp$ with respect to $\mathcal{T}$ and $\mathcal{V}$, similarly to the interference domain of Fig. 11, but also $\mathcal{C}$, as shown in Fig. 23. As $\mathcal{V}$, $\mathcal{T}$, and $\mathcal{C}$ are all finite, a map from $\mathcal{T} \times \mathcal{C} \times \mathcal{V}$ to $\mathcal{N}^\sharp$ can indeed be represented in memory, and the join $\cup^\sharp_\mathcal{I}$ and widening $\triangledown_\mathcal{I}$ can be computed pointwise. Moreover, abstract environments $\mathcal{E}^\sharp$ are also partitioned with respect

$$\mathcal{D}^\sharp_\mathcal{C} \stackrel{\text{def}}{=} (\mathcal{C} \to \mathcal{E}^\sharp) \times \mathcal{P}(\mathcal{L}) \times \mathcal{I}^\sharp$$
$$\gamma : \mathcal{D}^\sharp_\mathcal{C} \to \mathcal{D}_\mathcal{C}$$
$$\quad \text{s.t. } \gamma(R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} (\{\, (c, \rho) \mid c \in \mathcal{C},\ \rho \in \gamma_\mathcal{E}(R^\sharp(c)) \,\},\ \Omega,\ \gamma_\mathcal{I}(I^\sharp))$$
$$(R^\sharp_1, \Omega_1, I^\sharp_1) \cup^\sharp (R^\sharp_2, \Omega_2, I^\sharp_2) \stackrel{\text{def}}{=} (\lambda c : R^\sharp_1(c) \cup^\sharp_\mathcal{E} R^\sharp_2(c), \Omega_1 \cup \Omega_2, I^\sharp_1 \cup^\sharp_\mathcal{I} I^\sharp_2)$$
$$(R^\sharp_1, \Omega_1, I^\sharp_1) \triangledown (R^\sharp_2, \Omega_2, I^\sharp_2) \stackrel{\text{def}}{=} (\lambda c : R^\sharp_1(c) \triangledown_\mathcal{E} R^\sharp_2(c), \Omega_1 \cup \Omega_2, I^\sharp_1 \triangledown_\mathcal{I} I^\sharp_2)$$

Figure 24: Abstract domain of statements $\mathcal{D}^\sharp_\mathcal{C}$, derived from $\mathcal{E}^\sharp$ and $\mathcal{I}^\sharp$.

$$\underline{\mathbb{S}^\sharp_\mathcal{C}[\![\, s, t \,]\!] : \mathcal{D}^\sharp_\mathcal{C} \to \mathcal{D}^\sharp_\mathcal{C}}$$
$$\mathbb{S}^\sharp_\mathcal{C}[\![\, X \leftarrow e,\ t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=}$$
$$\quad \text{let } \forall c \in \mathcal{C} : (R^\sharp_c, \Omega_c) = \mathbb{S}^\sharp[\![\, X \leftarrow \mathit{apply}(t, c, R^\sharp, I^\sharp, e) \,]\!]\,(R^\sharp(c), \Omega) \text{ in}$$
$$\quad (\lambda c : R^\sharp_c,\ \textstyle\bigcup_{c \in \mathcal{C}} \Omega_c,\ I^\sharp[\forall c \in \mathcal{C} : (t, c, X) \mapsto I^\sharp(t, c, X) \cup^\sharp_\mathcal{N} \mathit{get}(X, R^\sharp_c)])$$
$$\mathbb{S}^\sharp_\mathcal{C}[\![\, e \bowtie 0?,\ t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=}$$
$$\quad \text{let } \forall c \in \mathcal{C} : (R^\sharp_c, \Omega_c) = \mathbb{S}^\sharp[\![\, \mathit{apply}(t, c, R^\sharp, I^\sharp, e) \bowtie 0? \,]\!]\,(R^\sharp(c), \Omega) \text{ in}$$
$$\quad (\lambda c : R^\sharp_c,\ \textstyle\bigcup_{c \in \mathcal{C}} \Omega_c,\ I^\sharp)$$
$$\mathbb{S}^\sharp_\mathcal{C}[\![\, \textbf{if } e \bowtie 0 \textbf{ then } s,\ t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=}$$
$$\quad (\mathbb{S}^\sharp_\mathcal{C}[\![\, s, t \,]\!] \circ \mathbb{S}^\sharp_\mathcal{C}[\![\, e \bowtie 0?,\ t \,]\!])(R^\sharp, \Omega, I^\sharp) \ \cup^\sharp\ \mathbb{S}^\sharp_\mathcal{C}[\![\, e \not\bowtie 0?,\ t \,]\!](R^\sharp, \Omega, I^\sharp)$$
$$\mathbb{S}^\sharp_\mathcal{C}[\![\, \textbf{while } e \bowtie 0 \textbf{ do } s,\ t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=}$$
$$\quad \mathbb{S}^\sharp_\mathcal{C}[\![\, e \not\bowtie 0?,\ t \,]\!](\mathit{lim}\ \lambda X^\sharp : X^\sharp \triangledown ((R^\sharp, \Omega, I^\sharp) \ \cup^\sharp\ (\mathbb{S}^\sharp_\mathcal{C}[\![\, s,\ t \,]\!] \circ \mathbb{S}^\sharp_\mathcal{C}[\![\, e \bowtie 0?,\ t \,]\!])X^\sharp))$$
$$\mathbb{S}^\sharp_\mathcal{C}[\![\, s_1;\ s_2,\ t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} (\mathbb{S}^\sharp_\mathcal{C}[\![\, s_2, t \,]\!] \circ \mathbb{S}^\sharp_\mathcal{C}[\![\, s_1, t \,]\!])(R^\sharp, \Omega, I^\sharp)$$

where:

$$\mathit{apply}(t, c, R^\sharp, I^\sharp, e) \stackrel{\text{def}}{=}$$
$$\quad \text{let } \forall Y \in \mathcal{V} : V^\sharp_Y = \cup^\sharp_\mathcal{N}\ \{\, I^\sharp(t', c', Y) \mid t' \neq t \wedge \mathit{intf}(c, c') \,\} \text{ in}$$
$$\quad \text{let } \forall Y \in \mathcal{V} : e_Y = \begin{cases} Y & \text{if } V^\sharp_Y = \bot^\sharp_\mathcal{N} \\ \mathit{as\text{-}expr}(V^\sharp_Y \cup^\sharp_\mathcal{N} \mathit{get}(Y, R^\sharp(c))) & \text{if } V^\sharp_Y \neq \bot^\sharp_\mathcal{N} \end{cases} \text{ in}$$
$$\quad e[\forall Y \in \mathcal{V} : Y \mapsto e_Y]$$

Figure 25: Abstract scheduled semantics of statements with interference.

to $\mathcal{C}$. Hence, the abstract semantic domain $\mathcal{D}^\sharp_\mathcal{C}$ abstracting $\mathcal{D}_\mathcal{C}$ (4.9) becomes:

$$\mathcal{D}^\sharp_\mathcal{C} \stackrel{\text{def}}{=} (\mathcal{C} \to \mathcal{E}^\sharp) \times \mathcal{P}(\mathcal{L}) \times \mathcal{I}^\sharp \ . \tag{4.11}$$

It is presented in Fig. 24.

Sound abstract transfer functions $\mathbb{S}^\sharp_\mathcal{C}$, derived from those in $\mathcal{E}^\sharp$ ($\mathbb{S}^\sharp$), are presented in Figs. 25–26.

Assignments and tests in Fig. 25 are very similar to the non-scheduled case $\mathbb{S}^\sharp_\mathcal{I}$ (Fig. 13) with two differences. Firstly, $\mathbb{S}^\sharp$ is applied pointwise to each abstract environment $R^\sharp(c) \in \mathcal{E}^\sharp$, $c \in \mathcal{C}$. New interferences due to assignments are also considered pointwise. Secondly, the *apply* function now takes as extra argument a configuration c, and then only considers interferences from configurations c' not in mutual exclusion with c. This is defined through

$$
\begin{array}{l}
\mathbb{S}^\sharp_{\mathcal{C}}[\![\, \textbf{lock}(m),\, t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} \\
\quad (\lambda(l,u,s) : \bigcup^\sharp_{\mathcal{E}} \ \{\, \mathit{in}^\sharp(t, l', \emptyset, m, R^\sharp(l', u', s), I^\sharp) \\
\qquad\qquad \mid l = l' \cup \{m\} \wedge u = \emptyset \wedge u' \subseteq \mathcal{M} \wedge s = \mathit{weak} \,\}, \\
\quad \Omega,\ I^\sharp \cup^\sharp_{\mathcal{I}} \bigcup^\sharp_{\mathcal{I}} \{\, \mathit{out}^\sharp(t, l, \emptyset, m', R^\sharp(l, u, s), I^\sharp) \mid l, u \subseteq \mathcal{M} \wedge m' \in u \wedge s = \mathit{weak} \,\}) \\
\mathbb{S}^\sharp_{\mathcal{C}}[\![\, \textbf{unlock}(m),\, t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} \\
\quad (\lambda(l,u,s) : \bigcup^\sharp_{\mathcal{E}} \{\, R^\sharp(l', u', s) \mid l = l' \setminus \{m\} \wedge u = u' \wedge s = \mathit{weak} \,\}, \\
\quad \Omega,\ I^\sharp \cup^\sharp_{\mathcal{I}} \bigcup^\sharp_{\mathcal{I}} \{\, \mathit{out}^\sharp(t, l \setminus \{m\}, u, m, R^\sharp(l, u, s), I^\sharp) \mid l, u \subseteq \mathcal{M} \wedge s = \mathit{weak} \,\}) \\
\mathbb{S}^\sharp_{\mathcal{C}}[\![\, \textbf{yield},\, t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} \\
\quad (\lambda(l,u,s) : \bigcup^\sharp_{\mathcal{E}} \{\, R^\sharp(l', u', s) \mid l = l' \wedge u = \emptyset \wedge u' \subseteq \mathcal{M} \wedge s = \mathit{weak} \,\}, \\
\quad \Omega,\ I^\sharp \cup^\sharp_{\mathcal{I}} \bigcup^\sharp_{\mathcal{I}} \{\, \mathit{out}^\sharp(t, l, \emptyset, m', R^\sharp(l, u, s), I^\sharp) \mid l, u \subseteq \mathcal{M} \wedge m' \in u \wedge s = \mathit{weak} \,\}) \\
\mathbb{S}^\sharp_{\mathcal{C}}[\![\, X \leftarrow \textbf{islocked}(m),\, t \,]\!](R^\sharp, \Omega, I^\sharp) \stackrel{\text{def}}{=} \\
\quad \text{let } (R^{\sharp\prime}, -, I^{\sharp\prime}) = \mathbb{S}^\sharp_{\mathcal{C}}[\![\, X \leftarrow [0,1],\, t \,]\!]\, (R^\sharp, \Omega, I^\sharp) \text{ in} \\
\quad \text{if no thread } t' > t \text{ locks } m, \text{ then:} \\
\qquad (\lambda(l,u,s) : \bigcup^\sharp_{\mathcal{E}} \ \{\, \text{let } (V^\sharp, -) = \\
\qquad\qquad \begin{cases} \mathbb{S}^\sharp[\![\, X \leftarrow 0 \,]\!](\mathit{in}^\sharp(t, l', u', m, R^\sharp(l', u', s), I^\sharp), \emptyset) & \text{if } m \in u \\ \mathbb{S}^\sharp[\![\, X \leftarrow 1 \,]\!](R^\sharp(l', u', s), \emptyset) & \text{if } m \notin u \end{cases} \\
\qquad\qquad \text{in } V^\sharp \\
\qquad\qquad \mid l = l' \wedge u \setminus \{m\} = u' \setminus \{m\} \wedge s = \mathit{weak} \,\}, \\
\qquad \Omega, I^{\sharp\prime}) \\
\quad \text{otherwise:} \\
\qquad (R^{\sharp\prime}, \Omega, I^{\sharp\prime})
\end{array}
$$

where:

$$
\begin{array}{l}
\mathit{in}^\sharp(t, l, u, m, V^\sharp, I^\sharp) \stackrel{\text{def}}{=} \\
\quad V^\sharp \cup^\sharp_{\mathcal{E}} \bigcup^\sharp_{\mathcal{E}} \ \{\, \text{let } X^\sharp = I^\sharp(t', (l', u', \mathit{sync}(m)), X) \text{ in} \\
\qquad\qquad \text{let } (V^{\sharp\prime}, -) = \mathbb{S}^\sharp[\![\, X \leftarrow \mathit{as\text{-}expr}(X^\sharp) \,]\!]\, (V^\sharp, \emptyset) \text{ in} \\
\qquad\qquad V^{\sharp\prime} \\
\qquad\qquad \mid X \in \mathcal{V} \wedge t \neq t' \wedge l \cap l' = l \cap u' = l' \cap u = \emptyset \,\} \\
\mathit{out}^\sharp(t, l, u, m, V^\sharp, I^\sharp) \stackrel{\text{def}}{=} \\
\quad \lambda(t', c, X) : \begin{cases} \mathit{get}(X, V^\sharp) & \text{if } t = t' \wedge c = (l, u, \mathit{sync}(m)), \\ & \exists c' = (l', -, \mathit{weak}) : m \in l' \wedge I^\sharp(t, c', X) \neq \bot^\sharp_{\mathcal{N}} \\ \bot^\sharp_{\mathcal{N}} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 26: Abstract scheduled semantics with interference of synchronization primitives.

the same function *intf* we used in the concrete semantics (Fig. 18). The semantics of non-primitive statements is the same as for previous semantics, by structural induction on the syntax of statements.

The semantics of synchronization primitives is presented in Fig. 26. It uses the functions $\mathit{in}^\sharp$ and $\mathit{out}^\sharp$ which abstract, respectively, the functions *in* and *out* presented in Fig. 19. As their concrete versions, $\mathit{in}^\sharp$ and $\mathit{out}^\sharp$ take as arguments a current thread $t \in \mathcal{T}$, a mutex $m \in \mathcal{M}$ protecting a critical section, and sets of mutexes $l, u \subseteq \mathcal{M}$ describing the current

scheduling configuration. Moreover, they take as arguments an abstract set of interferences $I^\sharp \in \mathcal{I}^\sharp$ instead of a concrete set, and an abstract set of environments $V^\sharp \in \mathcal{E}^\sharp$ instead of a single concrete one. The function $\mathit{out}^\sharp$ uses get (Fig. 10) to extract abstract sets of variable values from abstract environments and construct new abstract well synchronized interferences. The function $\mathit{in}^\sharp$ applies these interferences to an abstract environment by converting them to an expression (using $\mathit{as\text{-}expr}$) and updating the value of variables (using an assignment in $\mathbb{S}^\sharp$). Additionally, the semantics of synchronization primitives models updating the scheduler configuration from $c' = (l', u', \mathit{weak})$ to $c = (l, u, \mathit{weak})$ by moving abstract environments from $R^\sharp(c')$ into $R^\sharp(c)$; when partitions are collapsed, all the abstract environments mapped to the same configuration c are merged into $R^\sharp(c)$ using $\cup^\sharp_{\mathcal{E}}$. Finally, the abstract analysis $\mathbb{P}^\sharp_{\mathcal{C}}$ computes a fixpoint with widening over abstract interferences, which is similar to (3.6):

$$\begin{array}{l} \mathbb{P}^\sharp_{\mathcal{C}} \stackrel{\text{def}}{=} \Omega, \text{ where } (\Omega, -) \stackrel{\text{def}}{=} \\ \quad \mathit{lim}\, \lambda(\Omega, I^\sharp) : \text{let } \forall t \in \mathcal{T} : (-, \Omega'_t, I^{\sharp\prime}_t) = \mathbb{S}^\sharp_{\mathcal{C}}[\![\, \mathit{body}_t,\, t \,]\!]\, (R^\sharp_0, \Omega, I^\sharp) \text{ in} \\ \quad\quad (\bigcup \{\, \Omega'_t \mid t \in \mathcal{T} \,\},\ I^\sharp \triangledown_{\mathcal{I}} \bigcup^\sharp_{\mathcal{I}} \{\, I^{\sharp\prime}_t \mid t \in \mathcal{T} \,\}) \end{array} \tag{4.12}$$

where the partitioned initial abstract environment $R^\sharp_0 \in \mathcal{C} \to \mathcal{E}^\sharp$ is defined as:

$$R^\sharp_0 \stackrel{\text{def}}{=} \lambda c : \begin{cases} \mathcal{E}^\sharp_0 & \text{if } c = (\emptyset, \emptyset, \mathit{weak}) \\ \bot^\sharp_{\mathcal{E}} & \text{otherwise} \end{cases}$$

The resulting analysis is sound:

Theorem 4.2. $\mathbb{P}_{\mathcal{C}} \subseteq \mathbb{P}^\sharp_{\mathcal{C}}$.

Proof. In Appendix A.9. □

Due to partitioning, $\mathbb{P}^\sharp_{\mathcal{C}}$ is less efficient than $\mathbb{P}^\sharp_{\mathcal{I}}$. The abstract semantic functions for primitive statements, as well as the join $\cup^\sharp$ and widening $\triangledown$, are performed pointwise on all configurations $c \in \mathcal{C}$. However, a clever implementation need not represent explicitly nor iterate over partitions mapping a configuration to an empty environment $\bot^\sharp_{\mathcal{E}}$ or an empty interference $\bot^\sharp_{\mathcal{N}}$. The extra cost with respect to a non-scheduled analysis has thus a component that is linear in the number of non-$\bot^\sharp_{\mathcal{E}}$ environment partitions and a component linear in the number of non-$\bot^\sharp_{\mathcal{N}}$ interferences. Thankfully, partitioned environments are extremely sparse: Sec. 5 shows that, in practice, at most program points, $R^\sharp(c) = \bot^\sharp_{\mathcal{E}}$ except for a few configurations (at most 4 in our benchmark). Partitioned interferences are less sparse (52 in our benchmark) because, being flow-insensitive, they accumulate information for configurations reachable from any program point. However, this is not problematic: as interferences are non-relational, a larger number of partitions can be stored and manipulated efficiently.

Thanks to partitioning, the precision of $\mathbb{P}^\sharp_{\mathcal{C}}$ is much better than that of $\mathbb{P}^\sharp_{\mathcal{I}}$ in the presence of locks and priorities. For instance, $\mathbb{P}^\sharp_{\mathcal{C}}$ using the interval domain discovers that $T = 0$ in Fig. 14, while the analysis of Sec. 3.3 would only discover that $T \in [-1, 1]$ due to spurious interferences from the high priority thread.

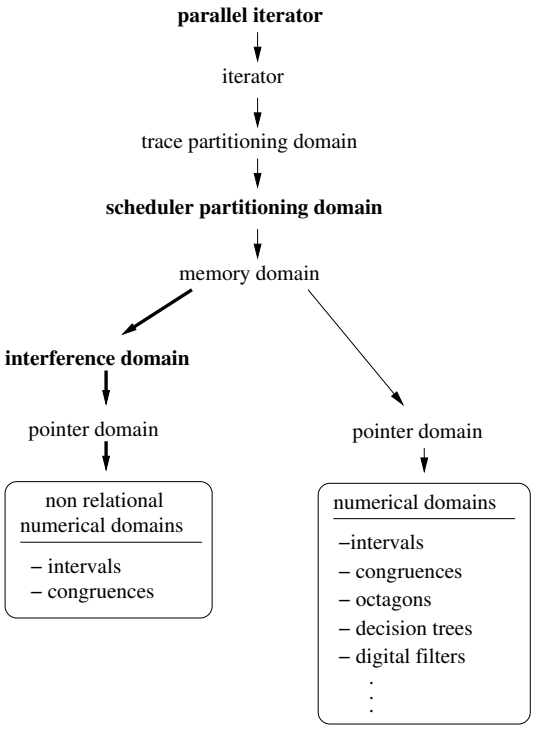


Figure 27: Hierarchy of abstractions in Astrée and Thésée. Domains in boldface are specific to Thésée and not included in Astrée.

5. Experimental Results

We implemented the abstract analysis of Sec. 4.5 in Thésée, our prototype analyzer based on the Astrée static analyzer [8]. We first describe succinctly Astrée, then Thésée, and finally our target application and its analysis by Thésée.

5.1. **The Astrée Analyzer.** Astrée is a static analyzer that checks for run-time errors in embedded C programs. Astrée accepts a fairly large subset of C, excluding notably dynamic memory allocation and recursion, that are generally unused (or even forbidden) in embedded code. Moreover, Astrée does not analyze multi-threaded programs, which is the very issue we address in the present article.

The syntax and semantics assumed by Astrée are based on the C99 norm [35], supplemented with the IEEE 754-1985 norm for floating-point arithmetics [33]. The C99 norm underspecifies many aspects of the semantics, leaving much leeway to compiler implementations, including random undocumented and unpredictable behaviors in case of an error such as an integer overflow. A strictly conforming program would rely only on the semantics defined in the norm. Few programs are strictly conforming; they rely instead on additional, platform-specific semantic hypotheses. This is especially true in the embedded software industry, where programs are designed for a specific, well-controlled platform, and not for portability. Thus, Astrée provides options to set platform-specific semantic features, such as the bit-size and byte-ordering of data-types, and the subsequent analysis is only sound with respect to these hypotheses. The run-time errors checked by Astrée are: overflows in

integer arithmetics and casts, integer divisions by zero, invalid bit-shifts, infinities and Not a Number floating-point values (caused by overflows, divisions by zero, or invalid operations), out-of-bound array accesses, invalid pointer arithmetics or dereferences (including null, dangling, and misaligned pointers), and failure of user-defined assertions (specified in a syntax similar to the standard `assert` function).

Astrée takes as input the C source after preprocessing by a standard preprocessor and a configuration file describing the ranges of the program inputs (such as memory-mapped sensors) if any. It then runs fully automatically and outputs a list of alarms corresponding to potential errors, and optionally program invariants for selected program points and variables. Astrée is sound in that it computes an over-approximation of all program traces, for all input scenarios. Moreover, the analysis continues even for erroneous program traces if the behavior after the error has a reasonable semantics. This is the case after integer overflows, for instance, using a wrap-around semantics, but it is not the case after dereferencing a dangling pointer, which has truly unpredictable results. In all cases, when there is no alarm, or when all the alarms can be proved by other means to be spurious, then the program is indeed proved free of run-time error.

Although Astrée accepts a large class of C programs, it cannot analyze most of them precisely and efficiently. It is specialized, by its choice of abstractions, towards control / command aerospace code, for which it gives good results. Thanks to a modular design, it can be adapted to other application domains by adding new abstractions. Actually, the initial specialization towards control / command avionic software [8] was achieved by incrementally adding new domains and refining existing ones until all false alarms could be removed on a target family of large control software from Airbus (up to 1 M lines) [23]. The resulting analyzer achieved the zero false alarm goal in a few hours of computation on a standard 2.66 GHz 64-bit intel server, and could be deployed in an industrial context [23]. This specialization can be continued with limited effort, at least for related application domains, as shown by our case study on space software [9]. Astrée is now a mature tool industrialized by AbsInt [1].

Figure 27 presents the design of Astrée as a hierarchy of abstract domains — we ignore for now boldface domains, which are specific to Thésée. Actually, Astrée does not contain a single "super-domain" but rather many small or medium-sized domains that focus on a specific kind of properties each, possess a specific encoding of these properties and algorithms to manipulate them, and can be easily plugged in and out. One of the first domain included in Astrée was the simple interval domain [13] that expresses properties of the form $X \in [a, b]$ for every machine integer and floating-point variable $X \in \mathcal{V}$. The interval domain is key as it is scalable, hence it can be applied to all variables at all program points. Moreover, it is able to express sufficient conditions for the absence of many kinds of errors, e.g., overflows. Astrée also includes relational domains, such as the octagon domain [46] able to infer relations of the form $\pm X \pm Y \leq c$. Such relations are necessary at a few locations, for instance to infer precise loop invariants, which then lead to tighter variable bounds. However, as the octagon domain is less scalable, it is used only on a few variables, selected automatically by a syntactic heuristic. Astrée also includes abstract domains specific to the target application domain, such as a domain to handle digital filtering featured in many control / command applications [27]. The computations are performed in all the domains in parallel, and the domains communicate information through a partially reduced product [18], so that they can improve each other in a controlled way — a fully reduced product, where all domains communicate all their finds, would not scale up. Additionally to numeric

variables, the C language features pointers. Pointer values are modeled in the concrete semantics of Astrée as semi-symbolic pairs containing a variable name and an integer byte-offset. The pointer abstract domain is actually a functor that adds support for pointers to any (reduced product of) numerical abstract domain(s) by maintaining internally for each pointer a set of pointed-to variables, and delegating the abstraction of the offset to the underlying numerical domain(s) (associating a synthetic integer variable to each offset). Another functor, the memory domain, handles the decomposition of aggregate variables (such as arrays and structures) into simpler scalar variables. The decomposition is dynamic to account for the weak type system of C and the frequent reinterpretation of the same memory portions as values of different types (due to union types and to type-punning). Both functors are described in [45]. Finally, a trace partitioning domain [44] adds a limited amount (for efficiency) of path-sensitivity by maintaining at the current control point several abstract states coming from execution traces with a different history (such as which branches of **if** statements were taken in the past). The computation in these domains is driven by an iterator that traverses the code by structural induction on its syntax, iterating loops with widening and stepping into functions to achieve a fully flow- and context-sensitive analysis.

More information and pointers about Astrée can be found in [7].

5.2. **The Thésée Analyzer.** Thésée is a prototype extension of Astrée that uses the abstract scheduled interference semantics of Sec. 4.5 to support the analysis of multi-threaded programs. Thésée checks for the same classes of run-time errors as Astrée. Additionally, it reports data-races, but ignores other parallel-related hazards, such as dead-locks and priority inversions, that are not described in our concrete semantics.

Thésée benefited directly from Astrée's numerous abstract domains and iteration strategies targeting embedded C code. Figure 27 presents the design of Thésée, where non-boldface domains are inherited from Astrée and boldface ones have been added.

Firstly, the memory domain has been modified to compute the abstract interferences generated by the currently analyzed thread and apply the interferences from other threads. We use the method of Fig. 25: the memory domain dynamically modifies expressions to include interferences explicitly (e.g., replacing variables with intervals) before passing the expressions to a stack of domains that are unaware of interferences. Interferences are themselves stored and manipulated by a specific domain which maintains abstract sets of values. Non-relational abstractions from Astrée, such as intervals but also abstract pointer values, are directly exploited to represent abstract interferences.

Secondly, a scheduler partitioning domain has been added. It maintains an abstraction of environments and of interferences for each abstract scheduled configuration live at the current program point. Then, for each configuration, it calls the underlying domain with the abstract environment associated to this configuration, as well as the abstract interferences that can effect this environment (i.e., a join of interferences from all configurations not in mutual exclusion with the current one). Additionally, the scheduler domain interprets directly all the instructions related to synchronization, which involves copying and joining abstract environments from different configurations, as described in Fig. 26.

Finally, we introduced an additional, parallel iterator driving the whole analysis. Following the execution model of the ARINC 653 specification, the parallel iterator first executes the main function as a regular single-threaded program and collects the set of resources (threads, synchronization objects) it creates. Then, as the program enters parallel mode,

the iterator analyzes each thread in sequence, and keeps re-analyzing them until their interferences are stable (in parallel mode, no new thread may be created).

All these changes add up to approximately 10 K lines of code in the 100 K lines analyzer and did not require much structural change.

5.3. **Analyzed Application.** Thésée has been applied to the analysis of a large industrial program consisting of 1.7 M lines of unpreprocessed C code[6] and 15 threads, and running under an ARINC 653 real-time OS [3]. The analyzed program is quite complex as it mixes string formatting, list sorting, network protocols (e.g., TFTP), and automatically generated synchronous logic.

The application performs system calls that must be properly modeled by the analyzer. To keep the analyzer simple, Thésée implements natively only low-level primitives to declare and manipulate threads as well as simple mutexes having the semantics described in Sec. 4.1. However, ARINC 653 objects have a more complex semantics. The analyzed program is thus completed with a 2,500-line hand-written model of the ARINC 653 standard, designed specifically for the analysis with Thésée. It implements all the system calls in C extended with Thésée primitives. The model maps high-level ARINC 653 objects to lower-level Thésée ones. For instance, ARINC processes[7] have a name while Thésée threads only have an integer identifier, so, the model keeps track of the correspondence between names and identifiers in C arrays and implements system calls to look up names and identifiers. It also emulates the ARINC semantics using Thésée primitives. For instance, a lock with a timeout is modeled as a non-deterministic test that either actually locks the mutex, or yields and returns an error code without locking the mutex. An important feature of the program we analyze is that all potentially blocking calls have a finite timeout, so, by construction, no dead-lock nor unbounded priority inversion can occur. This explains why we did not focus on detecting statically these issues in the present article.

5.4. **Analysis Results.** At the time of writing, the analysis with Thésée of this application takes 27 h on our 2.66 GHz 64-bit intel server. An important result is that only 5 iterations are required to stabilize abstract interferences. Moreover, there is a maximum of 52 partitions for abstract interferences and 4 partitions at most for abstract environments, so that the analysis fits in the 32 GB of memory of our server. The analysis currently generates 2,136 alarms (slightly less than one alarm per 800 lines of unpreprocessed code).

These figures have evolved before and during the writing of this article, as we improved the analysis. Figure 28 presents the evolution of the number of alarms on a period of 18 months. As our improvement effort focuses on optimizing the analysis precision, we do not present the detailed evolution of the analysis time (it oscillates between 14 h and 28 h,[8] with a number of iterations between 4 and 7) nor the memory consumption (stable at a little under 30 GB). The number of alarms started at 12,257 alarms mid-2010, as reported in [7, § VI]. This high initial number can be explained by the lack of specialization of the analyzer:

[6]After preprocessing and removal of comments, empty lines, and multiple definitions, the code is 2.1 M lines. The increase in size is due to the use of macros.

[7]In the ARINC 653 [3] terminology, execution units in shared memory are called "processes"; they correspond to POSIX threads and not to POSIX processes [34].

[8]Intuitively, adding domains and refining their precision degrades the efficiency. However, inferring tighter invariants can also reduce the number of loop iterations to reach a fixpoint, and so, improving the precision may actually lower the overall analysis time.

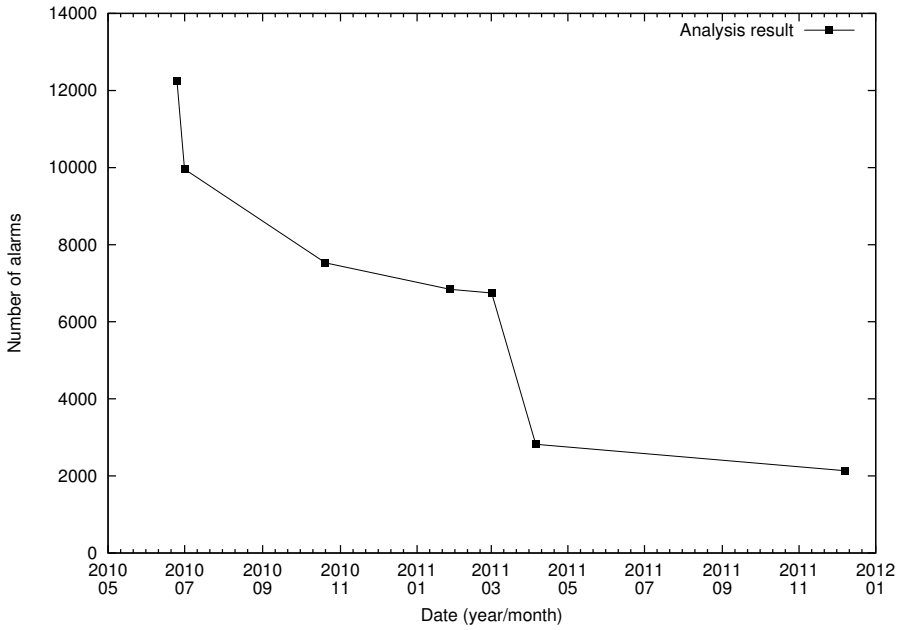


Figure 28: Evolution of the number of alarms in the analysis of our target application as we improved our prototype analyzer.

```
int clock, acc;

void accum(int reset)
{
  static int t0;
  if (reset) {
    acc = 0;
  }
  else {
    acc += clock - t0;
  }
  t0 = clock;
  /* 0 ≤ acc ≤ clock */
}
```

(a)

```
struct t {
  int id;
  struct { char msg[23]; } x[3];
} tab[12];

char* end_of_msg(int ident, int pos)
{
  int i;
  struct t* p = tab;
  for (i=0; i<12 && p[i].id!=ident; i++);
  char* m = p[i].x[pos].msg;
  for (i=0; i<23 && m[i]; i++);
  /* offset(m + i) ∈ 4 + 292[0, 11] + 96[0, 2] + [0, 22] */
  return m+i;
}
```

(b)

Figure 29: Program fragments that required an improvement in the analyzer prototype.

the first versions of Thésée were naturally tuned for avionic control / command software as they inherited abstract domains $\mathcal{E}^\sharp$ and $\mathcal{N}^\sharp$ from Astrée, but our target application for Thésée is *not* limited to control / command processing. To achieve our current results, we improved the numerical, pointer, and memory domains in Thésée, and designed new ones. We illustrate two of these improvements in Fig. 29. A first example is the improvement of transfer functions of existing domains. For instance, the function `accum` from Fig. 29.(a)

accumulates, in `acc`, elapsed time as counted by a `clock` variable (updated elsewhere), and we need to discover the invariant `acc` $\leq$ `clock`. This requires analyzing precisely the incrementation of `acc` in a relational domain. As the octagon domain we use [46] only supported precisely assignments involving two variables, we added new transfer functions for three-variable assignments (another solution, using the more powerful polyhedron domain [21], did not prove scalable enough). A second example is an improvement of the pointer domain to precisely track pointers traversing nested arrays and structures, as in Fig. 29.(b) where the precise location of the pointer `m+i` needs to be returned. Our target application features similar complex data-structures and their traversal extensively. We thus added a non-relational integer domain for offsets of the form $\alpha_0 + \sum_i \alpha_i[\ell_i, h_i]$, where the values of α_i, ℓ_i, and h_i are inferred dynamically. Note that all these improvements concern only the abstract domain parameters; neither the interference iterator nor the scheduler partitioning were refined.

Following the design-by-refinement used in Astrée [8], we have focused on the analysis of a single (albeit large and complex) industrial software and started refining the analyzer to lower the number of alarms, instead of multiplying shallower case studies. We plan to further improve the analysis precision in order to approach the zero false alarm goal. This is the objective of the AstréeA project [20], successor to Thésée. The remaining 2,136 alarms can be categorized into three kinds. Firstly, some alarms are, similarly to the ones described in Fig. 29, not related to parallelism but to the imprecision of the parameter abstract domains. An important class of properties currently not supported by Astrée nor Thésée is that of memory shape properties [12]. In the context of embedded software, dynamic memory allocation is disabled; nevertheless, our target code features dynamic linked lists allocated in large static arrays. Another class of properties concerns the correct manipulation of zero-terminated C strings. A significant part of the remaining alarms may be removed by designing new memory domains for these properties. Secondly, some alarms can be explained by an imprecise abstraction of thread interferences, similar to the imprecision observed in Figs. 21–22 (these examples were inspired from our target code). Hence the need to extend our framework to support relational and flow-sensitive abstractions of interferences. Thirdly, some alarms have simply not yet been fully investigated. Although Thésée provides verbose information on the context of each alarm as well as the thread-local and interference invariants, discovering the origin of alarms is a challenging task on such a large code: it often requires tracking the imprecision upstream and understanding the interplay of thread interferences.

6. Related Work

There are far too many works on the semantics and analysis of parallel programs to provide a fair survey and comparison here. Instead, we focus on a few works that are either recent or provide a fruitful comparison with ours.

6.1. **Interferences.** The idea of attaching to each thread location a local invariant and handling proofs of parallel programs similarly to that of sequential programs dates back to the Hoare-style logic of Owicki and Gries [49] and the inductive assertion method of Lamport [37, 40]. It has been well studied since; see [22] for a recent account and survey. The difference between the proofs of sequential programs and that of parallel programs in this framework is that the local invariants of each thread must be proved invariant by the

execution of all other threads — i.e., a non-interference check. These proof methods are studied from an Abstract Interpretation theoretical point of view by Cousot and Cousot [15], which leads to two results: an expression of each method as a decomposition of the global invariant into thread-local invariants, and a framework to apply abstractions and derive effective and sound static analyzers. When shifting from proof methods to inference methods, the non-interference check naturally becomes an inference of interferences. Our work is thus strongly inspired from [15]: it is based on an Owicki–Gries style decomposition of the global invariant (although it is not only based on the control points of threads, but also on a more complex scheduler state). The thread-local and interference parts are then abstracted separately, using a relational flow-sensitive analysis for the former and a coarser non-relational flow-insensitive analysis for the later. Our work is also similar to the recent static analysis of C programs with POSIX threads by Carré and Hymans [11]: both are based on Abstract Interpretation and interference computation, and both are implemented by modifying existing static analyses of sequential programs. Their analysis is more powerful than ours in that it handles dynamic thread creation and the concrete semantics models interferences as relations (instead of actions), but the subsequent abstraction leads to a non-relational analysis; moreover, real-time scheduling is not considered.

6.2. **Data-Flow Analysis.** Fully flow-insensitive analyses, such as Steensgaard's points-to analysis [58], naturally handle parallel programs. To our knowledge, all such analyses are also non-relational. These fast analyses are adequate for compiler optimization but, unfortunately, the level of accuracy required to prove safety properties demands the use of (at least partially) flow-sensitive and relational methods, which we do. By contrast, Sălcianu and Rinard [54] proposed a flow-sensitive pointer and escape analysis for parallel programs which is more precise (and more costly), although it still targets program optimisation. It uses a notion of interference to model the effect of threads and method calls.

6.3. **Model Checking.** Model-checking also has a long history of verifying parallel systems, including recently weak memory models [6]. To prevent state explosion, Godefroid [31] introduced partial order reduction methods. They limit the number of interleavings to consider, with no impact on soundness nor completeness. Due to the emphasis on completeness, the remaining set of interleavings can still be high. By contrast, we abstract the problem sufficiently so that no interleaving need to be considered at all, at the cost of completeness. Another way to reduce the complexity of model checking is the context bound approach, as proposed by Qadeer et al. [51]. As it is unsound, it may fail to find some run-time errors. By contrast, our method takes into account all executions until completion. In his PhD, Malkis [42] used abstract interpretation to prove the equivalence of Owicki and Gries's proof method and the more recent model-checking algorithm by Flanagan and Qadeer [29], and presented an improvement based on counterexample-guided abstract refinement, a method which, unlike ours, is not guaranteed to converge in finite time.

6.4. **Weakly Consistent Memories.** Weakly consistent memory models have been studied originally for hardware — see [2] for a tutorial. Precise formal models are now available for popular architectures, such as the intel x86 model by Sewell et al. [57], either inferred from informal processor documentations or reverse-engineered through "black-box" testing

[5]. Pugh [50] pioneered the use of weakly consistent memory models in programming language semantics in order to take into account hardware and compiler optimizations. This culminated in the Java memory model of Manson et al. [43, 32].

Weakly consistent memory models are now recognised as an important part of language semantics and are increasingly supported by verification methods. An example in model-checking is the work of Atig et al. [6]. An example in theorem proving is the extension by Ševčík et al. [56] of Leroy's formally proved C compiler [41]. Testing methods have also been proposed, such as that of Alglave et al. [5]. In the realm of static analysis, we can cite the static analysis by Abstract Interpretation of the happens-before memory model (at the core of the Java model) designed by Ferrara [28]. Recently, Alglave and al. proposed in [4] to lift analyses that are only sound with respect to sequential consistency, to analyses that are also sound in weak memory models. Their method is generic and uses a "repair loop" similar to our fixpoint of flow-insensitive interferences.

Memory models are often defined implicitly, by restricting execution traces using global conditions, following the approach chosen by the Java memory model [32]. We chose instead a generative model based on local control path transformations, which is reminiscent of the approach by Saraswat et al. [55]. We believe that it matches more closely classic software and hardware optimizations. Note that we focus on models that are not only realistic, but also amenable to abstraction into an interference semantics. The first condition ensures the soundness of the static analysis, while the second one ensures its efficiency.

6.5. **Real-Time Scheduling.** Many analyses of parallel programs assume arbitrary preemption, either implicitly at all program points (as in flow-insensitive analyses), or explicitly at specified program points (as in context-bounded approaches [51]), but few analyses model and exploit the strict scheduling policy of real-time schedulers. A notable exception is the work of Gamatié et al. [30] on the modeling of systems under an ARINC 653 operating system. As the considered systems are written in the SIGNAL language, their ARINC 653 model is naturally also written in SIGNAL, while ours in written in C (extended with low-level primitives for parallelism, which were not necessary when modeling in SIGNAL as the language can naturally express parallelism).

6.6. **Further Comparison.** A detailed comparison between domain-aware static analyzers, such as Astrée, and other verification methods, such as theorem proving and model checking, is presented in [19]. These arguments are still valid in the context of a parallel program analysis and not repeated here. On the more specific topic of parallel program analysis, we refer the reader to the comprehensive survey by Rinard [53].

7. Conclusion

We presented a static analysis by Abstract Interpretation to detect in a sound way run-time errors in embedded C software featuring several threads, a shared memory with weak consistency, mutual exclusion locks, thread priorities, and a real-time scheduler. Our method is based on a notion of interferences and a partitioning with respect to an abstraction of the scheduler state. It can be implemented on top of existing analyzers for sequential programs, leveraging a growing library of abstract domains. Promising early experimental results on an industrial code demonstrate the scalability of our approach.

A broad avenue for future work is to bridge the gap between the interleaving semantics and its incomplete abstraction using interferences. In particular, it seems important to abstract interferences due to well synchronized accesses in a relational way (this is in particular needed to remove some alarms remaining in our target application). We also wish to add some support for abstractions that are (at least partially) sensitive to the history of thread interleavings. This would be useful to exploit more properties of real-time schedulers, related for instance to the guaranteed ordering of some computations (by contrast, we focused in this article mainly on properties related to mutual exclusion).

Moreover, we wish to extend our framework to include more models of parallel computations. This includes support for alternate real-time operating systems with similar scheduling policies but manipulating different synchronization objects, for instance the condition variables in real-time POSIX systems [34], or alternate priority schemes, such as the priority ceiling protocol for mutexes. Another example is the support for the OSEK/VDX and Autosar real-time embedded platforms widely used in the automotive industry. We also wish to study more closely weak memory consistency semantics and, in particular, how to design more precise or more general interference semantics, and abstract them efficiently. Supporting atomic variables, recently included in the C and C++ languages, may also trigger the need for a finer, field-sensitive handling of weak memory consistency.

A long term goal is the analysis of other errors specifically related to parallelism, such as dead-locks, live-locks, and priority inversions. In a real-time system, all system calls generally have a timeout in order to respect hard deadlines. Thus, interesting properties are actually quantitative: by construction, unbounded priority inversions cannot occur, so, we wish to detect bounded priority inversions.

On the practical side, we wish to improve our prototype analyzer to reduce the number of false alarms on our target industrial code. This requires some of the improvements to the parallel analysis framework proposed above (such as relational and flow-sensitive abstractions for interferences), but also the design of new numerical, pointer, and memory domains which are not specific to parallel programs.

Acknowledgement

We wish to thank the ESOP'11 and LMCS anonymous reviewers as well as David Pichardie for their helpful comments on several versions of this article.

References

[1] AbsInt, Angewandte Informatik. Astrée run-time error analyzer. http://www.absint.com/astree.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Comp.*, 29(12):66–76, 1996.

[3] Aeronautical Radio Inc. ARINC 653. http://www.arinc.com.

[4] J. Alglave, D. Kroening, J. Lugton, V. Nimal, and M. Tautschnig. Soundness of data flow analyses for weak memory models. In *Proc. of the 9th Asian Symp. on Programming Languages and Systems (APLAS'2011)*, volume 7078 of *LNCS*, pages 272–288, Dec. 2011.

[5] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *Proc. of 17th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 41–44. Springer, Mar. 2011.

[6] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *Proc. of the 37th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages (POPL'10)*, pages 7–18. ACM, Jan. 2010.

[7] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385 in AIAA, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), Apr. 2010.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of the ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'03)*, pages 196–207. ACM, June 2003.

[9] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin. Space software validation using abstract interpretation. In *Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA'09)*, volume SP-669, pages 1–7. ESA, May 2009.

[10] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications (FMPA'93)*, volume 735 of *LNCS*, pages 128–141. Springer, June 1993.

[11] J.-L. Carré and C. Hymans. From single-thread to multithreaded: An efficient static analysis algorithm. Technical Report arXiv:0910.5833v1, EADS, Oct. 2009.

[12] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *Proc. of the 35th ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'08)*, pages 247–260. ACM, 2008.

[13] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, Jan. 1977.

[14] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[15] P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, NY, USA, 1984.

[16] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[17] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In *Int. Workshop on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.

[18] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In *Proc. of the 11th Annual Asian Computing Science Conf. (ASIAN'06)*, volume 4435 of *LNCS*, pages 272–300. Springer, Dec. 2006.

[19] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with Astrée, invited paper. In *Proc. of the First IEEE & IFIP Int. Symp. on Theoretical Aspects of Software Engineering (TASE'07)*, pages 3–17. IEEE CS Press, June 2007.

[20] P. Cousot, R. Cousot, J. Feret, A. Miné, and X. Rival. The AstréeA static analyzer. http://www.astreea.ens.fr.

[21] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the 5th Annual ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

[22] W.-P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[23] D. Delmas and J. Souyris. Astrée: from research to industry. In *Proc. of the 14th Int. Static Analysis Symp. (SAS'07)*, volume 4634 of *LNCS*, pages 437–451. Springer, Aug. 2007.

[24] E. W. Dijkstra. Cooperating sequential processes. In *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[25] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.

[26] J. Feret. Occurrence counting analysis for the pi-calculus. *Electronic Notes in Theoretical Computer Science*, 39(2), 2001.

[27] J. Feret. Static analysis of digital filters. In *Proc. of the 13th European Symp. on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 33–48. Springer, Mar. 2004.
[28] P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In *Proc. of the 2nd Int. Conf. on Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 116–133. Springer, 2008.
[29] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proc. of the 10th Int. SPIN Workshop on Model Checking of Software (SPIN'03)*, volume 2648 of *LNCS*, pages 213–224. Springer, 2003.
[30] A. Gamatié and T. Gautier. Synchronous modeling of avionics applications using the SIGNAL language. In *Proc. of the 9th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS'03)*, pages 144–151. IEEE Computer Society, 2003.
[31] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, Computer Science Department, 1994.
[32] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, June 2005.
[33] IEEE Computer Society. Standard for binary floating-point arithmetic. Technical report, ANSI/IEEE Std. 745-1985, 1985.
[34] IEEE Computer Society and The Open Group. Portable operating system interface (POSIX) – Application program interface (API) amendment 2: Threads extension (C language). Technical report, ANSI/IEEE Std. 1003.1c-1995, 1995.
[35] ISO/IEC JTC1/SC22/WG14 working group. C standard. Technical Report 1124, ISO & IEC, 2007.
[36] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 661–667. Springer, June 2009.
[37] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, 3(2):125–143, Mar. 1977.
[38] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.
[39] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In *IEEE Trans. on Computers*, volume 28, pages 690–691. IEEE Comp. Soc., Sep. 1979.
[40] L. Lamport. The "Hoare logic" of concurrent programs. *Acta Informatica*, 14(1):21–37, June 1980.
[41] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proc. of the 33rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'06)*, pages 42–54. ACM, 2006.
[42] A. Malkis. *Cartesian Abstraction and Verification of Multithreaded Programs*. PhD thesis, University of Freiburg, 2010.
[43] J. Manson, B. Pugh, and S. V. Adve. The Java memory model. In *Proc. of the 32nd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'05)*, pages 378–391. ACM, Jan. 2005.
[44] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In *Proc. of the 14th European Symp. on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 5–20. Springer, Apr. 2005.
[45] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63. ACM, June 2006.
[46] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
[47] A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *Proc. of the 20th European Symp. on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 398–418. Springer, Mar. 2011.
[48] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, Oct. 1999.
[49] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, Dec. 1976.
[50] B. Pugh. Fixing the Java memory model. In *Proc. of the ACM Conf. on Java Grande*, pages 89–98. ACM, 1999.

[51] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proc. of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.
[52] J. C. Reynolds. Toward a grainless semantics for shared-variable concurrency. In *Proc. of the Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *LNCS*, pages 35–48. Springer, Dec. 2004.
[53] M. C. Rinard. Analysis of multithreaded programs. In *Proc. of the 8th Int. Static Analysis Symp. (SAS'01)*, volume 2126 of *LNCS*, pages 1–19. Springer, Jul 2001.
[54] A. Sălcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proc. the 8th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming (PPoPP'01)*, pages 12–23. ACM, 2001.
[55] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programs (PPoPP'07)*, pages 161–172. ACM, Mar. 2007.
[56] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, P. Sewell, and S. Jagannathan. Relaxed-memory concurrency and verified compilation. In *Proc. of the 38th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages (POPL'11)*, pages 43–54. ACM, Jan. 2011.
[57] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Comm. ACM*, 53, 2010.
[58] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. of the 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'96)*, pages 32–41. ACM, 1996.
[59] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.
[60] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Proc. of the 26th IEEE/AIAA Digital Avionics Systems Conf. (DASC'07)*, volume 2.A.1, pages 1–10. IEEE, Oct. 2007.

Appendix A. Proof of Theorems

A.1. **Proof of Theorem 2.1.** $\forall s \in \mathit{stat} : \mathbb{S}[\![s]\!]$ is well defined and a complete $\sqcup-$morphism.

Proof. We prove both properties at the same time by structural induction on s.

- The case of assignments $X \leftarrow e$ and guards $e \bowtie 0?$ is straightforward.
- The semantics of conditionals and sequences is well-defined as its components are well-defined by induction hypothesis. It is a complete $\sqcup-$morphism, by induction hypothesis and the fact that the composition and the join of complete $\sqcup-$morphisms are complete $\sqcup-$morphisms.
- For a loop **while** $e \bowtie 0$ **do** s, consider F and G defined as:

$$F(X) \stackrel{\text{def}}{=} (\mathbb{S}[\![s]\!] \circ \mathbb{S}[\![e \bowtie 0?]\!])X$$
$$G(X) \stackrel{\text{def}}{=} (R, \Omega) \sqcup F(X) \ .$$

  By induction hypothesis, F, and so G, are complete $\sqcup-$morphisms. Thus, G has a least fixpoint [14]. We note that:

$$\mathbb{S}[\![\textbf{while } e \bowtie 0 \textbf{ do } s]\!](R, \Omega) = \mathbb{S}[\![e \not\bowtie 0?]\!](\mathit{lfp}\, G)$$

  which proves that the semantics of loops is well-defined. Moreover, according to [14] again, the least fixpoint can be computed by countable Kleene iterations: $\mathit{lfp}\, G =$

$\bigsqcup_{i\in\mathbb{N}} G^i(\emptyset,\emptyset)$. We now prove by induction on i that $G^i(\emptyset,\emptyset) = \bigsqcup_{k<i} F^k(R,\Omega)$. Indeed:

$$\begin{array}{rl} & G^1(\emptyset,\emptyset) \\ = & (R,\Omega) \sqcup F(\emptyset,\emptyset) \\ = & (R,\Omega) \\ = & F^0(R,\Omega) \end{array}$$

and

$$\begin{array}{rl} & G^{i+1}(\emptyset,\emptyset) \\ = & G\left(\bigsqcup_{k<i} F^k(R,\Omega)\right) \\ = & (R,\Omega) \sqcup F\left(\bigsqcup_{k<i} F^k(R,\Omega)\right) \\ = & (R,\Omega) \sqcup \bigsqcup_{k<i} F^{k+1}(R,\Omega) \\ = & \bigsqcup_{k<i+1} F^k(R,\Omega) \end{array}$$

because F is a $\sqcup-$morphism. As a consequence, $\mathit{lfp}\, G = \bigsqcup_{i\in\mathbb{N}} F^i(R,\Omega)$. Note that, $\forall i \in \mathbb{N} : F^i$ is a complete $\sqcup-$morphism. Thus, the function $(R,\Omega) \mapsto \mathit{lfp}\, G$ is also a complete $\sqcup-$morphism, and so is $\mathbb{S}[\![\, \textbf{while}\ e \bowtie 0\ \textbf{do}\ s\,]\!]$. □

A.2. **Proof of Theorem 2.2.** $\mathbb{P} \subseteq \mathbb{P}^\sharp$.

Proof. We prove by structural induction on s that:

$$\forall s, R^\sharp, \Omega : (\mathbb{S}[\![\, s\,]\!] \circ \gamma)(R^\sharp,\Omega) \sqsubseteq (\gamma \circ \mathbb{S}^\sharp[\![\, s\,]\!])(R^\sharp,\Omega)\ .$$

- The case of primitive statements $X \leftarrow e$ and $e \bowtie 0?$ holds by hypothesis: the primitive abstract functions provided by the abstract domain are assumed to be sound.
- The case of sequences $s_1; s_2$ is settled by noting that the composition of sound abstractions is a sound abstraction.
- The case of conditionals $\textbf{if}\ e \bowtie 0\ \textbf{then}\ s$ is settled by noting additionally that $\cup^\sharp$ is a sound abstraction of $\sqcup$, as $\cup^\sharp_{\mathcal{E}}$ is a sound abstraction of $\cup$.
- We now treat the case of loops. By defining:

$$\begin{array}{rcl} F^\sharp(X) & \stackrel{\text{def}}{=} & (R^\sharp,\Omega) \cup^\sharp (\mathbb{S}^\sharp[\![\, s\,]\!] \circ \mathbb{S}^\sharp[\![\, e \bowtie 0?\,]\!])(X) \\ F(X) & \stackrel{\text{def}}{=} & (R,\Omega) \sqcup (\mathbb{S}[\![\, s\,]\!] \circ \mathbb{S}[\![\, e \bowtie 0?\,]\!])(X) \end{array}$$

we have

$$\begin{array}{rcl} \mathbb{S}^\sharp[\![\, \textbf{while}\ e \bowtie 0\ \textbf{do}\ s\,]\!](R^\sharp,\Omega) & = & \mathbb{S}^\sharp[\![\, e \not\bowtie 0?\,]\!](\mathit{lim}\ \lambda X^\sharp : X^\sharp \triangledown F^\sharp(X^\sharp)) \\ \mathbb{S}[\![\, \textbf{while}\ e \bowtie 0\ \textbf{do}\ s\,]\!](R,\Omega) & = & \mathbb{S}[\![\, e \not\bowtie 0?\,]\!](\mathit{lfp}\, F)\ . \end{array}$$

By induction hypothesis, soundness of $\cup^\sharp$ and composition of sound functions, $F^\sharp$ is a sound abstraction of F. Assume now that $(R^{\sharp\prime},\Omega')$ is the limit of iterating $\lambda X : X \triangledown F^\sharp(X)$ from $(\bot^\sharp_{\mathcal{E}},\emptyset)$. Then, it is a fixpoint of $\lambda X^\sharp : X^\sharp \triangledown F^\sharp(X^\sharp)$, hence $(R^{\sharp\prime},\Omega') = (R^{\sharp\prime},\Omega') \triangledown F^\sharp(R^{\sharp\prime},\Omega')$. Applying γ and the soundness of $\triangledown$ and $F^\sharp$, we get:

$$\begin{array}{rl} & \gamma(R^{\sharp\prime},\Omega') \\ = & \gamma((R^{\sharp\prime},\Omega') \triangledown F^\sharp(R^{\sharp\prime},\Omega')) \\ \sqsupseteq & \gamma(F^\sharp(R^{\sharp\prime},\Omega')) \\ \sqsupseteq & F(\gamma(R^{\sharp\prime},\Omega'))\ . \end{array}$$

Thus, $\gamma(R^{\sharp\prime}, \Omega')$ is a post-fixpoint of F. As a consequence, it over-approximates $\mathit{lfp}\, F$, which implies the soundness of the semantics of loops.

We now apply the property we proved to $s = \mathit{body}$, $R^\sharp = \mathcal{E}_0^\sharp$ and $\Omega = \emptyset$, and use the monotony of $\mathbb{S}[\![s]\!]$ and the fact that $\gamma(\mathcal{E}_0^\sharp) \supseteq \mathcal{E}_0$ to get:

$$\begin{array}{ll} & \gamma(\mathbb{S}^\sharp[\![\mathit{body}]\!](\mathcal{E}_0^\sharp, \emptyset)) \\ \supseteq & \mathbb{S}[\![\mathit{body}]\!](\gamma(\mathcal{E}_0^\sharp, \emptyset)) \\ \supseteq & \mathbb{S}[\![\mathit{body}]\!](\mathcal{E}_0, \emptyset) \end{array}$$

which implies $\mathbb{P}^\sharp \supseteq \mathbb{P}$. □

A.3. **Proof of Theorem 2.3.** $\forall s \in \mathit{stat} : \mathbb{\Pi}[\![\pi(s)]\!] = \mathbb{S}[\![s]\!]$.

Proof. The proof is by structural induction on s.

- The case of primitive statements is straightforward as $\pi(s) = \{ s \}$.
- For sequences, we use the induction hypothesis and the fact that $\mathbb{\Pi}$ is a morphism for path concatenation to get:

$$\begin{array}{ll} & \mathbb{S}[\![s_1; s_2]\!] \\ = & \mathbb{S}[\![s_2]\!] \circ \mathbb{S}[\![s_1]\!] \\ = & \mathbb{\Pi}[\![\pi(s_2)]\!] \circ \mathbb{\Pi}[\![\pi(s_1)]\!] \\ = & \mathbb{\Pi}[\![\pi(s_1) \cdot \pi(s_2)]\!] \\ = & \mathbb{\Pi}[\![\pi(s_1; s_2)]\!] \ . \end{array}$$

- For conditionals, we also use the fact that $\mathbb{\Pi}$ is a morphism for $\cup$:

$$\begin{array}{ll} & \mathbb{S}[\![\textbf{if } e \bowtie 0 \textbf{ then } s]\!](R, \Omega) \\ = & (\mathbb{S}[\![s]\!] \circ \mathbb{S}[\![e \bowtie 0?]\!])(R, \Omega) \sqcup \mathbb{S}[\![e \not\bowtie 0?]\!](R, \Omega) \\ = & (\mathbb{\Pi}[\![\pi(s)]\!] \circ \mathbb{\Pi}[\![\{ e \bowtie 0? \}]\!])(R, \Omega) \sqcup \mathbb{\Pi}[\![\{ e \not\bowtie 0? \}]\!](R, \Omega) \\ = & \mathbb{\Pi}[\![((e \bowtie 0?) \cdot \pi(s)) \cup \{ e \not\bowtie 0? \}]\!](R, \Omega) \\ = & \mathbb{\Pi}[\![\pi(\textbf{if } e \bowtie 0 \textbf{ then } s)]\!](R, \Omega) \ . \end{array}$$

- For loops **while** $e \bowtie 0$ **do** s, we define F and G as in proof A.1, i.e., $F(X) \stackrel{\text{def}}{=} (\mathbb{S}[\![s]\!] \circ \mathbb{S}[\![e \bowtie 0?]\!])X$ and $G(X) \stackrel{\text{def}}{=} (R, \Omega) \sqcup F(X)$. Recall then that $\mathit{lfp}\, G = \bigsqcup_{i \in \mathbb{N}} F^i(R, \Omega)$. By induction hypothesis and $\cdot-$morphism, we have:

$$\begin{array}{ll} & F^i \\ = & (\mathbb{S}[\![s]\!] \circ \mathbb{S}[\![e \bowtie 0?]\!])^i \\ = & (\mathbb{\Pi}[\![\{e \bowtie 0?\} \cdot \pi(s)]\!])^i \\ = & \mathbb{\Pi}[\![(\{e \bowtie 0?\} \cdot \pi(s))^i]\!] \ . \end{array}$$

Let us now define the set of paths $P \stackrel{\text{def}}{=} \mathit{lfp}\, \lambda X : \{\epsilon\} \cup (X \cdot \{e \bowtie 0?\} \cdot \pi(s))$. By [14], $P = \bigcup_{i \in \mathbb{N}} (\{e \bowtie 0?\} \cdot \pi(s))^i$. As a consequence:

$$\mathbb{\Pi}[\![P]\!] = \bigsqcup_{i \in \mathbb{N}} \mathbb{\Pi}[\![(\{e \bowtie 0?\} \cdot \pi(s))^i]\!] = \bigsqcup_{i \in \mathbb{N}} F^i$$

and $\mathbb{\Pi}[\![P]\!](R, \Omega) = \bigsqcup_{i \in \mathbb{N}} F^i(R, \Omega) = \mathit{lfp}\, G$.

Finally:

$$\begin{aligned} & \mathbb{S}[\![\textbf{while } e \bowtie 0 \textbf{ do } s]\!](R,\Omega) \\ = \; & \mathbb{S}[\![e \not\bowtie 0?]\!](\mathit{lfp}\, G) \\ = \; & (\mathbb{\Pi}[\![\{ e \not\bowtie 0? \}]\!] \circ \mathbb{\Pi}[\![P]\!])(R,\Omega) \\ = \; & \mathbb{\Pi}[\![P \cdot \{e \not\bowtie 0?\}]\!](R,\Omega) \\ = \; & \mathbb{\Pi}[\![\pi(\textbf{while } e \bowtie 0 \textbf{ do } s)]\!](R,\Omega) \end{aligned}$$

which concludes the proof. □

A.4. **Proof of Theorem 3.1.** $\forall t \in \mathcal{T}, s \in \mathit{stat} : \mathbb{\Pi}_{\mathcal{I}}[\![\pi(s),\, t]\!] = \mathbb{S}_{\mathcal{I}}[\![s,\, t]\!]$.

Proof. The proof A.3 of Thm. 2.3 only relies on the fact that the semantic functions $\mathbb{S}[\![s]\!]$ are complete $\sqcup$−morphisms. As the functions $\mathbb{S}_{\mathcal{I}}[\![s, t]\!]$ are complete $\sqcup_{\mathcal{I}}$−morphisms, the same proof holds. □

A.5. **Proof of Theorem 3.2.** $\mathbb{P}_* \subseteq \mathbb{P}_{\mathcal{I}}$.

Proof. To ease the proof, we will use the notations $R(X)$, $\Omega(X)$, $I(X)$ to denote the various components of a triplet $X = (R(X), \Omega(X), I(X))$ in the concrete domain $\mathcal{D}_{\mathcal{I}} = \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\mathcal{L}) \times \mathcal{P}(\mathcal{I})$, and $V(X)$, $\Omega(X)$ for pairs $X = (V(X), \Omega(X))$ in $\mathcal{P}(\mathbb{R}) \times \mathcal{P}(\mathcal{L})$ output by the semantics of expressions.

Let $(\Omega_{\mathcal{I}}, I_{\mathcal{I}})$ be the fixpoint computed in (3.4), i.e., $(\Omega_{\mathcal{I}}, I_{\mathcal{I}}) = \bigsqcup_{t \in \mathcal{T}} (\Omega'_t, I'_t)$ where $(-, \Omega'_t, I'_t) = \mathbb{S}_{\mathcal{I}}[\![\mathit{body}_t, t]\!](\mathcal{E}_0, \Omega_{\mathcal{I}}, I_{\mathcal{I}})$. Then, by definition, $\mathbb{P}_{\mathcal{I}} = \Omega_{\mathcal{I}}$. We first prove the following properties that compare respectively the set of errors and environments output by the interleaving semantics and the interference semantics of any path $p \in \pi_*$:

(i) $\quad \Omega(\mathbb{\Pi}_*[\![p]\!](\mathcal{E}_0, \emptyset)) \subseteq \bigcup_{t \in \mathcal{T}} \Omega(\mathbb{\Pi}_{\mathcal{I}}[\![\mathit{proj}_t(p),\, t]\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}}))$

(ii) $\quad \forall t \in \mathcal{T},\, \rho \in R(\mathbb{\Pi}_*[\![p]\!](\mathcal{E}_0, \emptyset)) : \exists \rho' \in R(\mathbb{\Pi}_{\mathcal{I}}[\![\mathit{proj}_t(p),\, t]\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}})) :$
$\forall X \in \mathcal{V} : (\rho(X) = \rho'(X) \vee \exists t' \neq t : (t', X, \rho(X)) \in I_{\mathcal{I}})$

The proof is by induction on the length of p. The case $p = \epsilon$ is straightforward: the Ω component is $\emptyset$ on both sides of (i), and we can take $\rho' = \rho$ in (ii) as the R component is $\mathcal{E}_0$ on both sides. Consider now $p = p' \cdot (s', t')$, i.e., p is a path p' followed by an assignment or guard s' from thread t'. Consider some $\rho \in R(\mathbb{\Pi}_*[\![p']\!](\mathcal{E}_0, \emptyset))$ and the expression e' appearing in s'. We can apply the (ii) recurrence hypothesis to p' and t' which returns some $\rho' \in R(\mathbb{\Pi}_{\mathcal{I}}[\![\mathit{proj}_{t'}(p'),\, t']\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}}))$. The fact that, for any $X \in \mathcal{V}$, either $\rho(X) = \rho'(X)$ or $\exists t'' \neq t' : (t'', X, \rho(X)) \in I_{\mathcal{I}}$ implies, given the definition of $\mathbb{E}_{\mathcal{I}}[\![e']\!]$ (Fig. 7), that:

(iii) $\quad \mathbb{E}[\![e']\!]\rho \sqsubseteq \mathbb{E}_{\mathcal{I}}[\![e']\!](t', \rho', I_{\mathcal{I}})$

When considering, in particular, the error component $\Omega(\mathbb{E}[\![e']\!]\rho)$, (iii) allows extending the recurrence hypothesis (i) on p' to also hold on p, i.e., executing s' adds more errors on the right-hand side of (i) than on the left-hand side.

To prove (ii) on p, we consider two cases, depending on the kind of statement s':

- Assume that s' is a guard, say $e' = 0?$ (the proof is identical for other guards $e' \bowtie 0?$). Take any $t \in \mathcal{T}$ and $\rho \in R(\mathbb{\Pi}_*[\![p]\!](\mathcal{E}_0, \emptyset))$. By definition of $\mathbb{S}[\![e' = 0?]\!]$, we have $\rho \in R(\mathbb{\Pi}_*[\![p']\!](\mathcal{E}_0, \emptyset))$ and $0 \in V(\mathbb{E}[\![e']\!]\rho)$. Take any ρ' that satisfies the recurrence hypothesis (ii) on p' for t and ρ. We prove that it also satisfies (ii) on p for t and ρ.

The case $t \neq t'$ is straightforward as, in this case, $\mathit{proj}_t(p) = \mathit{proj}_t(p')$.

When $t = t'$, then $\mathit{proj}_t(p) = \mathit{proj}_t(p') \cdot (e' = 0?)$. Recall property (iii): $\mathbb{E}[\![e']\!]\rho \sqsubseteq \mathbb{E}_{\mathcal{I}}[\![e']\!](t', \rho', I_{\mathcal{I}})$, which implies that $0 \in V(\mathbb{E}_{\mathcal{I}}[\![e']\!](t', \rho', I_{\mathcal{I}}))$, and so, we have $\rho' \in R(\mathbb{\Pi}_{\mathcal{I}}[\![\mathit{proj}_t(p), t']\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}}))$.

- Assume that s' is an assignment $X \leftarrow e'$. Take any $\rho \in R(\mathbb{\Pi}_*[\![p]\!](\mathcal{E}_0, \emptyset))$ and $t \in \mathcal{T}$. Then ρ has the form $\rho_0[X \mapsto \rho(X)]$ for some $\rho_0 \in R(\mathbb{\Pi}_*[\![p']\!](\mathcal{E}_0, \emptyset))$ and $\rho(X) \in V(\mathbb{E}[\![e']\!]\rho_0)$.

  We first prove that $(t', X, \rho(X)) \in I_{\mathcal{I}}$. Take ρ'_0 as defined by the recurrence hypothesis (ii) for t', p' and ρ_0. Then $\rho(X) \in V(\mathbb{E}[\![e']\!]\rho_0) \subseteq V(\mathbb{E}_{\mathcal{I}}[\![e']\!](t', \rho'_0, I_{\mathcal{I}}))$ by property (iii). By definition of $\mathbb{S}_{\mathcal{I}}[\![X \leftarrow e', t']\!]$ (Fig. 8), we have $(t', X, \rho(X)) \in I(\mathbb{S}_{\mathcal{I}}[\![X \leftarrow e', t']\!](\{\rho'_0\}, \emptyset, I_{\mathcal{I}}))$. We have $\rho'_0 \in R(\mathbb{\Pi}_{\mathcal{I}}[\![\mathit{proj}_{t'}(p'), t']\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}}))$ by definition. Because $\mathit{proj}_{t'}(p) = \mathit{proj}_{t'}(p') \cdot (X \leftarrow e') \in \pi(\mathit{body}_{t'})$, it follows that $(t', X, \rho(X)) \in I(\mathbb{\Pi}_{\mathcal{I}}[\![\pi(\mathit{body}_{t'}), t']\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}}))$.
  By Thm. 3.1, we then have $(t', X, \rho(X)) \in I(\mathbb{S}_{\mathcal{I}}[\![\mathit{body}_{t'}, t']\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}})) = I'_{t'}$. Thus, $(t', X, \rho(X)) \in I_{\mathcal{I}}$, as $I_{\mathcal{I}}$ satisfies $I_{\mathcal{I}} = \bigcup_{t \in \mathcal{T}} I'_t$.

  To prove (ii), we consider first the case where $t \neq t'$. Take ρ'_0 as defined by the recurrence hypothesis (ii) for t, p' and ρ_0. As $t \neq t'$, $\mathit{proj}_t(p) = \mathit{proj}_t(p')$, and $\rho'_0 \in R(\mathbb{\Pi}_{\mathcal{I}}[\![\mathit{proj}_t(p), t]\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}}))$. As ρ and ρ_0 are equal except maybe on X, and we have $(t', X', \rho(X)) \in I_{\mathcal{I}}$, then ρ'_0 satisfies (ii) for t, p, and ρ.

  We now consider the case where $t = t'$. Take ρ'_0 as defined by the recurrence hypothesis (ii) for t, p' and ρ_0. We define $\rho' = \rho'_0[X \mapsto \rho(X)]$. The property (iii) implies $V(\mathbb{E}[\![e']\!]\rho_0) \subseteq V(\mathbb{E}_{\mathcal{I}}[\![e']\!](t', \rho'_0, I_{\mathcal{I}}))$. We get $\rho' \in R(\mathbb{\Pi}_{\mathcal{I}}[\![\mathit{proj}_t(p') \cdot (X \leftarrow e'), t]\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}})) = R(\mathbb{\Pi}_{\mathcal{I}}[\![\mathit{proj}_t(p), t]\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}}))$. As ρ and ρ_0 are equal except maybe on X, and ρ' and ρ'_0 are also equal except maybe on X, and on X we have $\rho(X) = \rho'(X)$, then ρ' satisfies (ii) for t, p and ρ.

The theorem then stems from applying property (i) to all $p \in \pi_*$ and using Thm. 3.1:

$$
\begin{aligned}
& \mathbb{P}_* \\
= \; & \Omega(\mathbb{\Pi}_*[\![\pi_*]\!](\mathcal{E}_0, \emptyset)) \\
= \; & \textstyle\bigcup_{p \in \pi_*} \Omega(\mathbb{\Pi}_*[\![p]\!](\mathcal{E}_0, \emptyset)) \\
\subseteq \; & \textstyle\bigcup_{t \in \mathcal{T}, p \in \pi_*} \Omega(\mathbb{\Pi}_{\mathcal{I}}[\![\mathit{proj}_t(p), t]\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}})) \\
= \; & \textstyle\bigcup_{t \in \mathcal{T}} \Omega(\mathbb{\Pi}_{\mathcal{I}}[\![\pi(\mathit{body}_t), t]\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}})) \\
= \; & \textstyle\bigcup_{t \in \mathcal{T}} \Omega(\mathbb{S}_{\mathcal{I}}[\![\mathit{body}_t, t]\!](\mathcal{E}_0, \emptyset, I_{\mathcal{I}})) \\
\subseteq \; & \textstyle\bigcup_{t \in \mathcal{T}} \Omega(\mathbb{S}_{\mathcal{I}}[\![\mathit{body}_t, t]\!](\mathcal{E}_0, \Omega_{\mathcal{I}}, I_{\mathcal{I}})) \\
= \; & \Omega_{\mathcal{I}} \\
= \; & \mathbb{P}_{\mathcal{I}} \; .
\end{aligned}
$$

□

A.6. **Proof of Theorem 3.3.** $\mathbb{P}_{\mathcal{I}} \subseteq \mathbb{P}^\sharp_{\mathcal{I}}$.

Proof. We start by considering the semantics of expressions. We first note that, for any abstract environment $R^\sharp \in \mathcal{E}^\sharp$, abstract interference $I^\sharp \in \mathcal{I}^\sharp$, thread t, and expression e, if $\rho \in \gamma_{\mathcal{E}}(R^\sharp)$, then $\mathbb{E}_{\mathcal{I}}[\![e]\!](t, \rho, \gamma_{\mathcal{I}}(I^\sharp)) \subseteq \mathbb{E}[\![\mathit{apply}(t, R^\sharp, I^\sharp, e)]\!]\rho$, i.e., *apply* can be used to over-approximate the semantics with interference $\mathbb{E}_{\mathcal{I}}$ using the non-interfering semantics $\mathbb{E}$ in the concrete. We prove this by structural induction on the syntax of expressions e. We

consider first the case of variables $e = X \in \mathcal{V}$, which is the interesting case. When $\gamma_{\mathcal{E}}(R^\sharp)$ has no interference for X, then $\mathit{apply}(t, R^\sharp, I^\sharp, X) = X$ and:

$$\mathbb{E}[\![\ \mathit{apply}(t, R^\sharp, I^\sharp, X)\ ]\!]\rho = \mathbb{E}[\![\ X\ ]\!]\rho = \{\rho(X)\} = \mathbb{E}_{\mathcal{I}}[\![\ X\ ]\!](t, \rho, \gamma_{\mathcal{I}}(I^\sharp))\ .$$

In case of interferences on X, we have $\mathit{apply}(t, R^\sharp, I^\sharp, X) = \mathit{as\text{-}expr}(V^\sharp_X \cup^\sharp_{\mathcal{N}} \mathit{get}(X, R^\sharp))$, which is an interval $[l, h]$ containing, by definition of *as-expr* and soundness of *get*, both $\rho(X)$ and $\{\, v \mid \exists t' \neq t : (t', X, v) \in \gamma_{\mathcal{I}}(I^\sharp)\,\}$. Thus, $\mathbb{E}[\![\ \mathit{apply}(t, R^\sharp, I^\sharp, X)\ ]\!]\rho = [l, h] \subseteq \mathbb{E}[\![\ X\ ]\!]\rho$. For other expressions, we note that $\mathbb{E}_{\mathcal{I}}$ and $\mathbb{E}$ are defined in similar way, and so, the proof stems from the induction hypothesis and the monotony of $\mathbb{E}_{\mathcal{I}}$ and $\mathbb{E}$.

To prove the soundness of primitive statements $\mathbb{S}^\sharp_{\mathcal{I}}[\![\ s\ ]\!]$, we combine the above result with the soundness of $\mathbb{S}^\sharp$ with respect to $\mathbb{S}$. We immediately get that $R((\mathbb{S}_{\mathcal{I}}[\![\ s\ ]\!] \circ \gamma)(R^\sharp, \Omega, I^\sharp)) \subseteq R((\gamma \circ \mathbb{S}^\sharp_{\mathcal{I}}[\![\ s\ ]\!])(R^\sharp, \Omega, I^\sharp))$ for an assignment or test s, and likewise for the Ω component. (We reuse the notations $R(X)$, $\Omega(X)$, $I(X)$ from the proof A.5.) The I component is unchanged by $\mathbb{S}_{\mathcal{I}}[\![\ s\ ]\!]$ and $\mathbb{S}^\sharp_{\mathcal{I}}[\![\ s\ ]\!]$ when s is a test. For the I component after an assignment, we remark that $I((\mathbb{S}_{\mathcal{I}}[\![\ X \leftarrow e\ ]\!] \circ \gamma)(R^\sharp, \Omega, I))$ can be written as

$$\gamma_{\mathcal{I}}(I) \cup \{\, (t, X, \rho(X)) \mid \rho \in R((\mathbb{S}_{\mathcal{I}}[\![\ X \leftarrow e\ ]\!] \circ \gamma)(R^\sharp, \Omega, I))\,\}\ .$$

We then reuse the fact that $R((\mathbb{S}_{\mathcal{I}}[\![\ s\ ]\!] \circ \gamma)(R^\sharp, \Omega, I^\sharp)) \subseteq R((\gamma \circ \mathbb{S}^\sharp_{\mathcal{I}}[\![\ s\ ]\!])(R^\sharp, \Omega, I^\sharp))$ to conclude the proof of primitive statements.

The case of non-primitive statements is easier as it is mostly unchanged between $\mathbb{S}$ and $\mathbb{S}_{\mathcal{I}}$ (hence, between $\mathbb{S}^\sharp$ and $\mathbb{S}^\sharp_{\mathcal{I}}$). Hence, the proof in A.2 of Thm. 2.2 still applies.

As a consequence, we have:

$$\forall t \in \mathcal{T} : (\mathbb{S}_{\mathcal{I}}[\![\ \mathit{body}_t,\ t\ ]\!] \circ \gamma)(R^\sharp, \Omega, I^\sharp) \sqsubseteq_{\mathcal{I}} (\gamma \circ \mathbb{S}^\sharp_{\mathcal{I}}[\![\ \mathit{body}_t,\ t\ ]\!])(R^\sharp, \Omega, I^\sharp)\ .$$

Consider now the solutions $(\Omega_{\mathcal{I}}, I_{\mathcal{I}})$ and $(\Omega^\sharp_{\mathcal{I}}, I^\sharp_{\mathcal{I}})$ of the fixpoints (3.4) and (3.6). We have:

$$\begin{array}{rl} & (\Omega_{\mathcal{I}}, I_{\mathcal{I}}) = \mathit{lfp}\ F \\ \text{where} & F(\Omega, I) = \bigsqcup_{t \in \mathcal{T}} (\Omega'_t, I'_t) \\ \text{and} & (-, \Omega'_t, I'_t) = \mathbb{S}_{\mathcal{I}}[\![\ \mathit{body}_t, t\ ]\!](\mathcal{E}_0, \Omega, I) \end{array}$$

Likewise:

$$\begin{array}{rl} & (\Omega^\sharp_{\mathcal{I}}, I^\sharp_{\mathcal{I}}) = \mathit{lim}\ F^\sharp \\ \text{where} & F^\sharp(\Omega^\sharp, I^\sharp) = (\Omega^\sharp, I^\sharp) \triangledown \bigsqcup^\sharp_{t \in \mathcal{T}} (\Omega^{\sharp\prime}_t, I^{\sharp\prime}_t) \\ \text{and} & (-, \Omega^{\sharp\prime}_t, I^{\sharp\prime}_t) = \mathbb{S}^\sharp_{\mathcal{I}}[\![\ \mathit{body}_t,\ t\ ]\!](\mathcal{E}^\sharp_0, \Omega^\sharp, I^\sharp) \\ \text{defining} & (\Omega^\sharp_1, I^\sharp_1) \sqcup^\sharp (\Omega^\sharp_2, I^\sharp_2) \stackrel{\text{def}}{=} (\Omega^\sharp_1 \cup \Omega^\sharp_2, I^\sharp_1 \cup^\sharp_{\mathcal{I}} I^\sharp_2) \\ \text{and} & (\Omega^\sharp_1, I^\sharp_1) \triangledown (\Omega^\sharp_2, I^\sharp_2) \stackrel{\text{def}}{=} (\Omega^\sharp_2, I^\sharp_1 \triangledown_{\mathcal{I}} I^\sharp_2) \end{array}$$

By soundness of $\cup^\sharp_{\mathcal{I}}$, $\triangledown_{\mathcal{I}}$, $\mathbb{S}^\sharp_{\mathcal{I}}$, and $\mathcal{E}^\sharp_0$, we get that $F^\sharp$ is a sound abstraction of F. The same fixpoint transfer property that was used for loops in proof A.2 can be used here to prove that $\mathit{lim}\ F^\sharp$ is a sound abstraction of $\mathit{lfp}\ F$. As a consequence, we have $\Omega^\sharp_{\mathcal{I}} \supseteq \Omega_{\mathcal{I}}$, and so, $\mathbb{P}^\sharp_{\mathcal{I}} \supseteq \mathbb{P}_{\mathcal{I}}$. □

A.7. **Proof of Theorem 3.5.** $\mathbb{P}'_* \subseteq \mathbb{P}_{\mathcal{I}}$.

Proof. We reuse the notations $R(X)$, $\Omega(X)$, $I(X)$, $V(X)$ from proof A.5. Consider a path p from a thread t that gives, under an elementary transformation from Def. 3.4, a path p'. Let us denote by $\mathcal{V}_f$ the subset of fresh variables (i..e, that do not appear anywhere in the program, except maybe in p'), and by $\mathcal{V}_l(t)$ the subset of variables that are local to t (i.e., that do not appear anywhere, except maybe in thread t).

We consider triples (R, Ω, I) such that the interferences I are consistent with the fresh and local variables, i.e., if $(t, X, v) \in I$, then $X \notin \mathcal{V}_f$ and $X \in \mathcal{V}_l(t') \Longrightarrow t = t'$. We prove that, for such triples, the following holds:

$$\text{(i)} \quad \Omega(\mathbb{\Pi}_{\mathcal{I}}[\![\, p', t \,]\!](R, \Omega, I)) \subseteq \Omega(\mathbb{\Pi}_{\mathcal{I}}[\![\, p, t \,]\!](R, \Omega, I))$$

$$\text{(ii)} \quad \forall (t', X, v) \in I(\mathbb{\Pi}_{\mathcal{I}}[\![\, p', t \,]\!](R, \Omega, I)) : \\ (t', X, v) \in I(\mathbb{\Pi}_{\mathcal{I}}[\![\, p, t \,]\!](R, \Omega, I)) \text{ or } (t = t' \wedge X \in \mathcal{V}_l(t))$$

$$\text{(iii)} \quad \forall \rho' \in R(\mathbb{\Pi}_{\mathcal{I}}[\![\, p', t \,]\!](R, \Omega, I)) : \exists \rho \in R(\mathbb{\Pi}_{\mathcal{I}}[\![\, p, t \,]\!](R, \Omega, I)) : \\ \forall X \in \mathcal{V} : \rho(X) = \rho'(X) \text{ or } X \in \mathcal{V}_f$$

i.e., the transformation does not add any error (i), it can only add interferences on local variables (ii), and change environments on fresh variables (iii). We now prove (i)–(iii) for each case in Def. 3.4.

(1) Redundant store elimination: $X \leftarrow e_1 \cdot X \leftarrow e_2 \ \rightsquigarrow \ X \leftarrow e_2$, where $X \notin \mathit{var}(e_2)$ and $\mathit{nonblock}(e_1)$.
We note:
- $(R_i, \Omega_i, I_i) = \mathbb{\Pi}_{\mathcal{I}}[\![\, X \leftarrow e_i,\, t \,]\!](R, \Omega, I)$ for $i = 1, 2$, and
- $(R_{1;2}, \Omega_{1;2}, I_{1;2}) = \mathbb{\Pi}_{\mathcal{I}}[\![\, X \leftarrow e_1 \cdot X \leftarrow e_2,\, t \,]\!](R, \Omega, I)$.

As $X \notin \mathit{var}(e_2)$, $\mathbb{E}_{\mathcal{I}}[\![\, e_2 \,]\!](t, \rho, I) = \mathbb{E}_{\mathcal{I}}[\![\, e_2 \,]\!](t, \rho[X \mapsto v], I)$ for any ρ and v. Moreover, as $\mathit{nonblock}(e_1)$:

$$\forall \rho \in R : \exists v \in V(\mathbb{E}_{\mathcal{I}}[\![\, e_1 \,]\!](t, \rho, I)) : \rho[X \mapsto v] \in R_1 \ .$$

This implies $R_{1;2} = R_2$, and so, (iii). Moreover, $\Omega_{1;2} = \Omega_1 \cup \Omega_2 \supseteq \Omega_2$, and so, (i). Likewise, $I_{1;2} = I_1 \cup I_2 \supseteq I_2$, and so, (ii).

Note that the hypothesis $\mathit{nonblock}(e_1)$ is important. Otherwise we could allow $X \leftarrow 1/_{\ell}\, 0 \cdot X \leftarrow 1/_{\ell'}\, 0 \ \rightsquigarrow \ X \leftarrow 1/_{\ell'} 0$, where the error ℓ' is in the transformed program but not in the original one (here, $\Omega_{1;2} = \Omega_1 \not\supseteq \Omega_2$).

(2) Identity store elimination $X \leftarrow X \ \rightsquigarrow \ \epsilon$.
We have:

$$\begin{array}{lcl} \Omega(\mathbb{\Pi}_{\mathcal{I}}[\![\, X \leftarrow X,\, t \,]\!](R, \Omega, I)) & = & \Omega \\ I(\mathbb{\Pi}_{\mathcal{I}}[\![\, X \leftarrow X,\, t \,]\!](R, \Omega, I)) & \supseteq & I \\ R(\mathbb{\Pi}_{\mathcal{I}}[\![\, X \leftarrow X,\, t \,]\!](R, \Omega, I)) & \supseteq & R \ . \end{array}$$

In the two last inequalities, the converse inequality does not necessarily hold. Indeed, $X \leftarrow X$ creates interferences that may be observed by other thread. Moreover $V(\mathbb{E}_{\mathcal{I}}[\![\, X \,]\!](t, \rho, I))$ is not necessarily $\{\, \rho(X) \,\}$, but may contain interferences from other threads. This shows in particular that the converse transformation, identity insertion, is not acceptable as it introduces interferences.

(3) Reordering assignments: $X_1 \leftarrow e_1 \cdot X_2 \leftarrow e_2 \ \rightsquigarrow \ X_2 \leftarrow e_2 \cdot X_1 \leftarrow e_1$, where $X_1 \notin \mathit{var}(e_2)$, $X_2 \notin \mathit{var}(e_1)$, $X_1 \neq X_2$, and $\mathit{nonblock}(e_1)$.
We note:

– $(R_i, \Omega_i, I_i) = \mathbb{\Pi}_{\mathcal{I}}[\![X_i \leftarrow e_i,\, t]\!](R, \Omega, I)$ for $i = 1, 2$, and
– $(R_{1;2}, \Omega_{1;2}, I_{1;2}) = \mathbb{\Pi}_{\mathcal{I}}[\![X_1 \leftarrow e_1 \cdot X_2 \leftarrow e_2,\, t]\!](R, \Omega, I)$, and
– $(R_{2;1}, \Omega_{2;1}, I_{2;1}) = \mathbb{\Pi}_{\mathcal{I}}[\![X_2 \leftarrow e_2 \cdot X_1 \leftarrow e_1, t]\!](R, \Omega, I)$.

As $X_2 \notin \mathit{var}(e_1)$, $X_1 \notin \mathit{var}(e_2)$, and $X_1 \neq X_2$, we have

$$\forall \rho, v : \mathbb{E}_{\mathcal{I}}[\![e_1]\!](t, \rho, I) = \mathbb{E}_{\mathcal{I}}[\![e_1]\!](t, \rho[X_2 \mapsto v], I) \text{ and}$$
$$\forall \rho, v : \mathbb{E}_{\mathcal{I}}[\![e_2]\!](t, \rho, I) = \mathbb{E}_{\mathcal{I}}[\![e_2]\!](t, \rho[X_1 \mapsto v], I) \ .$$

This implies $R_{1;2} = R_{2;1}$, and so, (iii).

As $\mathit{nonblock}(e_1)$, we have $\forall \rho \in R : \exists v : \rho[X_1 \mapsto v] \in R_1$. This implies $\Omega_2 = \Omega(\mathbb{\Pi}_{\mathcal{I}}[\![X_2 \leftarrow e_2,\, t]\!](R_1, \Omega, I))$, and so $\Omega_{1;2} = \Omega_1 \cup \Omega_2$. Likewise, $I_{1;2} = I_1 \cup I_2$. Moreover, $\Omega_{2;1} \subseteq \Omega_1 \cup \Omega_2 = \Omega_{1;2}$ and $I_{2;1} \subseteq I_1 \cup I_2 = I_{1;2}$. Thus (i) and (ii) hold.

As in (1), the $\mathit{nonblock}(e_1)$ hypothesis is important so that errors in e_2 masked by $X_1 \leftarrow e_1$ in the original program do not appear in the transformed program.

(4) Reordering guards: $e_1 \bowtie 0? \cdot e_2 \bowtie' 0? \ \rightsquigarrow \ e_2 \bowtie' 0? \cdot e_1 \bowtie 0?$, where $\mathit{noerror}(e_2)$.
We use the same notations as in the preceding proof. We have $I_{2;1} = I_{1;2} = I$, which proves (ii). Consider $\rho \in R$, then either $\rho \in R_1 \cap R_2$, in which case $\rho \in R_{1;2}$ and $\rho \in R_{2;1}$, or $\rho \notin R_1 \cap R_2$, in which case $\rho \notin R_{1;2}$ and $\rho \notin R_{2;1}$. In both cases, $R_{1;2} = R_{2;1}$, which proves (iii). We have $\Omega_2 \subseteq \Omega_{2;1} \subseteq \Omega_1 \cup \Omega_2$ and $\Omega_1 \subseteq \Omega_{1;2} \subseteq \Omega_1 \cup \Omega_2$. But, as $\mathit{noerror}(e_2)$, $\Omega_2 = \emptyset$, which implies $\Omega_{2;1} \subseteq \Omega_1 \subseteq \Omega_{1;2}$, and so, (i).

The property $\mathit{noerror}(e_2)$ is important. Consider otherwise the case where $e_1 \bowtie 0?$ filters out an environment leading to an error in e_2. The error would appear in the transformed program but not in the original one.

(5) Reordering guards before assignments: $X_1 \leftarrow e_1 \cdot e_2 \bowtie 0? \ \rightsquigarrow \ e_2 \bowtie 0? \cdot X_1 \leftarrow e_1$, where $X_1 \notin \mathit{var}(e_2)$ and either $\mathit{nonblock}(e_1)$ or $\mathit{noerror}(e_2)$.
We use the same notations as in the preceding proofs. As $X_1 \notin \mathit{var}(e_2)$, we have

$$\forall \rho, v : \mathbb{E}_{\mathcal{I}}[\![e_2]\!](t, \rho, I) = \mathbb{E}_{\mathcal{I}}[\![e_2]\!](t, \rho[X_1 \mapsto v], I) \ .$$

This implies $R_{1;2} = R_{2;1}$, and so, (iii).

Moreover, we have $I_{1;2} = I_1$ and $I_{2;1} \subseteq I_1$, thus (ii) holds.

For (i), consider first the case $\mathit{nonblock}(e_1)$. We have $\forall \rho \in R : \exists v : \rho[X_1 \mapsto v] \in R_1$. This implies $\Omega_2 = \Omega(\mathbb{\Pi}_{\mathcal{I}}[\![e_2 \bowtie 0?,\, t]\!](R_1, \Omega, I))$, and so $\Omega_{1;2} = \Omega_1 \cup \Omega_2$. As $\Omega_2 \subseteq \Omega_{2;1} \subseteq \Omega_1 \cup \Omega_2$, (i) holds. Consider now the case $\mathit{noerror}(e_2)$. We have $\Omega_{1;2} = \Omega_1$ and $\Omega_{2;1} \subseteq \Omega_1$, and so, (i) holds.

(6) Reordering assignments before guards: $e_1 \bowtie 0? \cdot X_2 \leftarrow e_2 \ \rightsquigarrow \ X_2 \leftarrow e_2 \cdot e_1 \bowtie 0?$, where $X_2 \notin \mathit{var}(e_1)$, $X_2 \in \mathcal{V}_l(t)$, and $\mathit{noerror}(e_2)$.
As $X_2 \notin \mathit{var}(e_1)$, we have

$$\forall \rho, v : \mathbb{E}_{\mathcal{I}}[\![e_1]\!](t, \rho, I) = \mathbb{E}_{\mathcal{I}}[\![e_1]\!](t, \rho[X_2 \mapsto v], I)$$

and thus, using the same notations as before, $R_{1;2} = R_{2;1}$, and so, (iii).

Because, $\mathit{noerror}(e_2)$, we have $\Omega_{2;1} \subseteq \Omega_1 \cup \Omega_2 = \Omega_1$ and $\Omega_{1;2} = \Omega_1$, and so $\Omega_{2;1} \subseteq \Omega_{1;2}$ (i).

Unlike the preceding proofs, we now have $I_{1;2} \subseteq I_{2;1} = I_2$ and generally $I_{1;2} \not\supseteq I_{2;1}$. However, as $X_2 \in \mathcal{V}_l(t)$, we have $I_{2;1} \setminus I_{1;2} \subseteq \{t\} \times \mathcal{V}_l(t) \times \mathbb{R}$, i.e., the only interferences added by the transformation concern the variable X_2, local to t. This is sufficient to ensure (ii).

(7) Assignment propagation: $X \leftarrow e \cdot s \;\rightsquigarrow\; X \leftarrow e \cdot s[e/X]$, where $X \notin \mathit{var}(e)$, $\mathit{var}(e) \subseteq \mathcal{V}_l(t)$, and e is deterministic.
Let us note :
- $(R_1, \Omega_1, I_1) = \mathbb{\Pi}_{\mathcal{I}}[\![X \leftarrow e,\, t]\!](R, \Omega, I)$,
- $(R_{1;2}, \Omega_{1;2}, I_{1;2}) = \mathbb{\Pi}_{\mathcal{I}}[\![s,\, t]\!](R_1, \Omega_1, I_1)$,
- $(R'_{1;2}, \Omega'_{1;2}, I'_{1;2}) = \mathbb{\Pi}_{\mathcal{I}}[\![s[e/X],\, t]\!](R_1, \Omega_1, I_1)$.

Take $\rho_1 \in R_1$, then there exists $\rho \in R$ such that $\rho_1 = \rho[X \mapsto \rho_1(X)]$ and $\rho_1(X) \in V(\mathbb{E}_{\mathcal{I}}[\![e]\!](t, \rho, I))$. As e is deterministic and $X \notin \mathit{var}(e)$, $V(\mathbb{E}[\![X]\!]\rho_1) = V(\mathbb{E}[\![e]\!]\rho) = \{\rho_1(X)\}$. Additionally, as $\mathit{var}(e) \subseteq \mathcal{V}_l(t)$, there is no interference in I for variables in e from other threads, and so, $V(\mathbb{E}_{\mathcal{I}}[\![X]\!](t, \rho_1, I_1)) = V(\mathbb{E}_{\mathcal{I}}[\![e]\!](t, \rho, I)) = \{\rho_1(X)\}$. As a consequence, $R_{1;2} = R'_{1;2}$, which proves (iii). Moreover, $I_{1;2} = I'_{1;2}$, hence (ii) holds. Finally, we have $\bigcup \{\, \Omega(\mathbb{E}_{\mathcal{I}}[\![e]\!](t, \rho, I)) \mid \rho \in R \,\} \subseteq \Omega_1$ but also $\bigcup \{\, \Omega(\mathbb{E}_{\mathcal{I}}[\![e]\!](t, \rho_1, I_1)) \mid \rho_1 \in R_1 \,\} \subseteq \Omega_1$, i.e., any error from the evaluation of e while executing $s[e/X]$ in R_1 was already present during the evaluation of e while executing $X \leftarrow e$ in R. Thus, $\Omega_{1;2} = \Omega'_{1;2}$, and (i) holds.

The fact that e is deterministic is important. Otherwise, consider $X \leftarrow [0,1] \cdot Y \leftarrow X + X \;\rightsquigarrow\; X \leftarrow [0,1] \cdot Y \leftarrow [0,1] + [0,1]$. Then $Y \in \{0, 2\}$ on the left of $\rightsquigarrow$ while $Y \in \{0, 1, 2\}$ on the right of $\rightsquigarrow$: the transformed program has more behaviors than the original one. Likewise, $\mathit{var}(e) \subseteq \mathcal{V}_l(t)$ ensures that e may not evaluate to a different value due to interferences from other threads.

(8) Sub-expression elimination: $s_1 \cdot \ldots \cdot s_n \;\rightsquigarrow\; X \leftarrow e \cdot s_1[X/e] \cdot \ldots \cdot s_n[X/e]$, where $X \in \mathcal{V}_f$, $\mathit{var}(e) \cap \mathit{lval}(s_i) = \emptyset$, and $\mathit{noerror}(e)$.
Let us note:

$$(R', \Omega', I') = \mathbb{\Pi}_{\mathcal{I}}[\![X \leftarrow e,\, t]\!](R, \Omega, I) \;.$$

Consider $\rho' \in R'$. Then $\rho' = \rho[X \mapsto \rho'(X)]$ for some $\rho \in R$ and $\rho'(X) \in V(\mathbb{E}_{\mathcal{I}}[\![e]\!](t, \rho, I))$. As X does not appear in s_i (being fresh), and noting:

$$(R'_i, \Omega'_i, I'_i) = \mathbb{\Pi}_{\mathcal{I}}[\![s_1[X/e] \cdot \ldots \cdot s_i[X/e],\, t]\!](\{\, \rho' \,\}, \Omega', I')$$

we get:

$$\forall i,\, \rho'_i \in R'_i : V(\mathbb{E}[\![X]\!]\rho'_i) = \{\rho'(X)\}$$

and, as $X \in \mathcal{V}_f$, there is no interference on X, and

$$\forall \rho'_i \in R'_i : V(\mathbb{E}_{\mathcal{I}}[\![X]\!](t, \rho'_i, I'_i)) = \{\rho'(X)\} \;.$$

As $\mathit{var}(e) \cap \mathit{lval}(s_i) = \emptyset$, and noting

$$(R_i, \Omega_i, I_i) = \mathbb{\Pi}_{\mathcal{I}}[\![s_1 \cdot \ldots \cdot s_i,\, t]\!](\{\rho\}, \Omega, I)$$

we get:

$$\forall i,\, \rho_i \in R_i : V(\mathbb{E}_{\mathcal{I}}[\![e]\!](t, \rho_i, I_i)) = V(\mathbb{E}_{\mathcal{I}}[\![e]\!](t, \rho, I)) \supseteq \{\rho'(X)\} \;.$$

As a consequence, $\forall i,\, \rho'_i \in R'_i : \exists \rho_i \in R_i$ such that $\rho'_i = \rho_i[X \mapsto \rho'(X)]$. When $i = n$, this implies (iii). Another consequence is that $\forall i : I'_i \subseteq I_i \cup \{\, (t, X, \rho'(X)) \mid \rho' \in R' \,\}$. As $X \in \mathcal{V}_f$, this implies (ii) when $i = n$. Moreover, as $\mathit{noerror}(e)$, $\Omega' = \Omega$. Note that:

$$\forall i : \Omega(\mathbb{\Pi}_{\mathcal{I}}[\![s[X/e],\, t]\!](R'_i, \Omega'_i, I'_i)) \subseteq \Omega(\mathbb{\Pi}_{\mathcal{I}}[\![s,\, t]\!](R'_i, \Omega'_i, I'_i)) \;.$$

Thus, $\forall i : \Omega'_i \subseteq \Omega_i$, which implies (i) when $i = n$.

(9) Expression simplification: $s \rightsquigarrow s[e'/e]$, when $\forall \rho : \mathbb{E}[\![e]\!]\rho \sqsupseteq \mathbb{E}[\![e']\!]\rho$ and $\mathit{var}(e) \cup \mathit{var}(e') \subseteq \mathcal{V}_l(t)$.
As $\mathit{var}(e) \subseteq \mathcal{V}_l(t)$, there is no interference in I for variables in e from other threads. We deduce that $\mathbb{E}_\mathcal{I}[\![e]\!](t, \rho, I) = \mathbb{E}[\![e]\!]\rho$. Likewise, $\mathbb{E}_\mathcal{I}[\![e']\!](t, \rho, I) = \mathbb{E}[\![e']\!]\rho$, and so, $\mathbb{E}_\mathcal{I}[\![e]\!](t, \rho, I) \sqsupseteq \mathbb{E}_\mathcal{I}[\![e']\!](t, \rho, I)$. By monotony of $\mathbb{S}_\mathcal{I}$, we get:

$$\mathbb{\Pi}_\mathcal{I}[\![s,\, t]\!](R, \Omega, I) \sqsupseteq \mathbb{\Pi}_\mathcal{I}[\![s[e'/e],\, t]\!](R, \Omega, I)$$

which implies (i)–(iii).

We now prove that, if the pair p, p' satisfies (i)–(iii), then so does the pair $p{\cdot}s$, $p'{\cdot}s$ for any primitive statement s executed by t. We consider a triple (R_0, Ω_0, I_0) and note $(R', \Omega', I') = \mathbb{\Pi}_\mathcal{I}[\![p', t]\!](R_0, \Omega_0, I_0)$ and $(R, \Omega, I) = \mathbb{\Pi}_\mathcal{I}[\![p, t]\!](R_0, \Omega_0, I_0)$. Take any $\rho' \in R'$. By (iii) on the pair p, p', there is some $\rho \in R$ that equals ρ' except maybe for some variables that are free, and so, cannot occur in s. Moreover, by (ii) on p, p', I contains I' except maybe for some interferences that are from t, and so, cannot influence the expression in s. So, by noting e the expression appearing in s, $\mathbb{E}_\mathcal{I}[\![e]\!](t, \rho', I') \sqsubseteq_\mathcal{I} \mathbb{E}_\mathcal{I}[\![e]\!](t, \rho, I)$. As a consequence, (i) and (ii) hold for the pair $p{\cdot}s$, $p'{\cdot}s$. Consider now $\rho' \in R(\mathbb{\Pi}_\mathcal{I}[\![p' \cdot s,\, t]\!](R_0, \Omega_0, I_0))$. If s is a guard $e = 0?$ (the other guards $e \bowtie 0?$ being similar), then $\rho' \in R'$ and, by (iii) for p, p', $\exists \rho \in R$ equal to ρ' except for some free variables. As, $0 \in V(\mathbb{E}_\mathcal{I}[\![e]\!](t, \rho', I')) \subseteq V(\mathbb{E}_\mathcal{I}[\![e]\!](t, \rho, I))$, ρ proves the property (iii) for $p{\cdot}s$, $p'{\cdot}s$. If s is an assignment $X \leftarrow e$ then $\rho' = \rho'_0[X \mapsto \rho'(X)]$ for some $\rho'_0 \in R'$ and $\rho'(X) \in V(\mathbb{E}_\mathcal{I}[\![e]\!](t, \rho'_0, I'))$. By (iii) for p, p', $\exists \rho_0 \in R$ equal to ρ'_0 except for some free variables. So, $\rho'(X) \in V(\mathbb{E}_\mathcal{I}[\![e]\!](t, \rho'_0, I')) \subseteq V(\mathbb{E}_\mathcal{I}[\![e]\!](t, \rho_0, I))$. Thus, $\rho_0[X \mapsto \rho'(X)]$ proves the property (iii) for $p \cdot s$, $p' \cdot s$.

The following two properties are much easier to prove. Firstly, if the pair p, p' satisfies (i)–(iii), so does the pair $q \cdot p$, $q \cdot p'$ for any path q. This holds because (i)–(iii) are stated for all R, Ω and I. Secondly, if both pairs p, p' and p', p'' satisfy (i)–(iii), then so does the pair p, p''. This completes the proof that elementary path transformations can be applied in a context with an arbitrary prefix and suffix, and that several transformations can be applied sequentially.

We are now ready to prove the theorem. Consider the least fixpoint computed by the interference semantics (3.4):

$$\begin{aligned}
&(\Omega_\mathcal{I}, I_\mathcal{I}) = \mathit{lfp}\, F \\
&\text{where } F(\Omega, I) = (\textstyle\bigcup_{t\in\mathcal{T}} \Omega_t,\, \bigcup_{t\in\mathcal{T}} I_t) \\
&\text{and } (-, \Omega_t, I_t) = \mathbb{S}_\mathcal{I}[\![\ \mathit{body}_t,\, t]\!](\mathcal{E}_0, \Omega, I)\ .
\end{aligned}$$

By Thm. 3.1, we have $(-, \Omega_t, I_t) = \mathbb{\Pi}_\mathcal{I}[\![\pi(\mathit{body}_t),\, t]\!](\mathcal{E}_0, \Omega, I)$. Given transformed threads $\pi'(t)$, consider also the fixpoint:

$$\begin{aligned}
&(\Omega'_\mathcal{I}, I'_\mathcal{I}) = \mathit{lfp}\, F' \\
&\text{where } F'(\Omega', I') = (\textstyle\bigcup_{t\in\mathcal{T}} \Omega'_t,\, \bigcup_{t\in\mathcal{T}} I'_t \setminus \mathcal{I}_l) \\
&\text{and } (-, \Omega'_t, I'_t) = \mathbb{\Pi}_\mathcal{I}[\![\pi'(t),\, t]\!](\mathcal{E}_0, \Omega', I')
\end{aligned}$$

and $\mathcal{I}_l$ is a set of interferences we can ignore as they only affect local or fresh variables:

$$\mathcal{I}_l = \{\, (t, X, v) \mid t \in \mathcal{T},\, v \in \mathbb{R},\, X \in \mathcal{V}_f \cup \mathcal{V}_l(t) \,\}\ .$$

Then, given each path in $p' \in \pi'(t)$, we can apply properties (i) and (ii) to the pair p, p', where p is the path in $\pi(\mathit{body}_t)$ that gives p' after transformation. We get that $F'(X) \sqsubseteq_\mathcal{I} F(X)$. As a consequence, $\mathit{lfp}\, F' \sqsubseteq_\mathcal{I} \mathit{lfp}\, F$. The transformed semantics, however, is not

exactly $\mathit{lfp}\, F'$, but rather:

$$\begin{array}{l}(\Omega''_t, I''_t) = \mathit{lfp}\, F'' \\ \text{where } F''(\Omega', I') = (\bigcup_{t\in\mathcal{T}} \Omega'_t, \bigcup_{t\in\mathcal{T}} I'_t) \\ \text{and } (\Omega'_t, I'_t) \text{ defined as before}\end{array}$$

The difference lies in the extra interferences generated by the transformed program, which are all in $\mathcal{I}_l$. Such interferences, however, have no influence on the semantics of threads, as we have:

$$\mathbb{\Pi}_{\mathcal{I}}[\![\, \pi'(t),\, t \,]\!](\mathcal{E}_0, \Omega, I') = \mathbb{\Pi}_{\mathcal{I}}[\![\, \pi'(t),\, t \,]\!](\mathcal{E}_0, \Omega, I' \setminus \mathcal{I}_l) \ .$$

Indeed, any interference $(t', X, v) \in \mathcal{I}_l$ is either from thread t and then ignored for the thread t itself, or it is from another thread $t' \neq t$ in which case X is a local variable of t or a free variable, which does not occur in t'. As a consequence, $I'_t = I''_t$ and $\Omega''_t = \Omega'_t$, and so, $\Omega'' = \Omega' \subseteq \Omega$. Hence, the interference semantics of the original program contains all the errors that can occur in any program obtained by acceptable thread transformations. □

A.8. **Proof of Theorem 4.1.** $\mathbb{P}_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$ and $\mathbb{P}'_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$.

Proof. We first prove $\mathbb{P}_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$.

In order to do so, we need to consider a path-based interference semantics $\mathbb{\Pi}_{\mathcal{C}}[\![\, P, t \,]\!]$ that, given a set of paths $P \subseteq \pi(\mathit{body}_t)$ in a thread t, computes:

$$\mathbb{\Pi}_{\mathcal{C}}[\![\, P, t \,]\!](R, \Omega, I) \stackrel{\text{def}}{=} \bigsqcup\nolimits_{\mathcal{C}} \{\, (\mathbb{S}_{\mathcal{C}}[\![\, s_n, t \,]\!] \circ \cdots \circ \mathbb{S}_{\mathcal{C}}[\![\, s_1, t \,]\!])(R, \Omega, I) \mid s_1 \cdot \ldots \cdot s_n \in P \,\} \ .$$

Similarly to Thms. 2.3, 3.1, the two semantics are equal:

$$\forall t \in \mathcal{T},\, s \in \mathit{stat} : \mathbb{\Pi}_{\mathcal{C}}[\![\, \pi(s),\, t \,]\!] = \mathbb{S}_{\mathcal{C}}[\![\, s,\, t \,]\!]$$

The proof is identical to that in A.3 as the $\mathbb{S}_{\mathcal{C}}$ functions are complete $\sqcup$−morphisms, and so, we do not repeat it here.

The rest of the proof that $\mathbb{P}_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$ follows a similar structure as A.5. Let $(\Omega_{\mathcal{C}}, I_{\mathcal{C}})$ be the fixpoint computed in (4.10), i.e.,

$$\begin{array}{l}(\Omega_{\mathcal{C}}, I_{\mathcal{C}}) = \bigsqcup\nolimits_{\mathcal{C}} \{\, (\Omega'_t, I'_t) \mid t \in \mathcal{T} \,\} \\ \text{where } (-, \Omega'_t, I'_t) = \mathbb{S}_{\mathcal{C}}[\![\, \mathit{body}_t, t \,]\!](\{c_0\} \times \mathcal{E}_0, \Omega_{\mathcal{C}}, I_{\mathcal{C}}) \ .\end{array}$$

We denote initial states for $\mathbb{P}_{\mathcal{H}}$ and $\mathbb{P}_{\mathcal{C}}$ as respectively $\mathcal{E}_{0h} \stackrel{\text{def}}{=} \{h_0\} \times \mathcal{E}_0$ and $\mathcal{E}_{0c} \stackrel{\text{def}}{=} \{c_0\} \times \mathcal{E}_0$. Furthermore, we link scheduler states $h \in \mathcal{H}$ and partitioning configurations $c \in \mathcal{C}$ in a thread t with the following abstraction $\alpha_t : \mathcal{H} \to \mathcal{P}(\mathcal{C})$:

$$\alpha_t(b, l) \stackrel{\text{def}}{=} \{\, (l(t), u, \mathit{weak}) \mid \forall x \in u, t' \in \mathcal{T} : x \notin l(t') \,\} \ .$$

i.e., a configuration forgets the ready state $b(t)$ of the thread (*ready*, *yield*ing, or *wait*ing for some mutex), but remembers the exact set of mutexes $l(t)$ held by the current thread, and optionally remembers the mutexes not held by any thread u. We prove the following properties by induction on the length of the path $p \in \pi_*$:

(i) $\Omega(\mathbb{\Pi}_{\mathcal{H}}[\![\, p \,]\!](\mathcal{E}_{0h}, \emptyset)) \subseteq \bigcup_{t\in\mathcal{T}} \Omega(\mathbb{\Pi}_{\mathcal{C}}[\![\, \mathit{proj}_t(p),\, t \,]\!](\mathcal{E}_{0c}, \emptyset, I_{\mathcal{C}}))$

(ii) $\forall t \in \mathcal{T} : \forall (h, \rho) \in R(\mathbb{\Pi}_{\mathcal{H}}[\![\, p \,]\!](\mathcal{E}_{0h}, \emptyset)) : \exists (c, \rho') \in R(\mathbb{\Pi}_{\mathcal{C}}[\![\, \mathit{proj}_t(p),\, t \,]\!](\mathcal{E}_{0c}, \emptyset, I_{\mathcal{C}})) :$
$c \in \alpha_t(h) \wedge$
$\forall X \in \mathcal{V} : \rho(X) = \rho'(X)$ or $\exists t', c' : t \neq t',\ \mathit{intf}(c, c'),\ (t', c', X, \rho(X)) \in I_{\mathcal{C}}$.

The properties hold for $p = \epsilon$ as $\forall t \in \mathcal{T}$:

$$\begin{array}{lcl} \mathbb{\Pi}_{\mathcal{H}}[\![\, \epsilon \,]\!](\mathcal{E}_{0h}, \emptyset) & = & (\mathcal{E}_{0h}, \emptyset) \\ \mathbb{\Pi}_{\mathcal{C}}[\![\, \epsilon, t \,]\!](\mathcal{E}_{0c}, \emptyset, I_{\mathcal{C}}) & = & (\mathcal{E}_{0c}, \emptyset, I_{\mathcal{C}}) \end{array}$$

and $c_0 \in \alpha_t(h_0)$.

Assume that the properties hold for p' and consider $p = p' \cdot (s', t')$, i.e., p' followed by a primitive statement s' executed by a thread t'. The case where s' is an assignment or a guard is very similar to that of proof A.5: we took care to update (ii) to reflect the change in the evaluation of variables in expressions $\mathbb{E}_{\mathcal{C}}[\![\, X \,]\!]$ (in particular, the use of *intf* to determine which interferences from other threads can influence the current one, given their respective configuration). The effect of $\mathit{enabled}_t$ in $\mathbb{S}_{\mathcal{H}}[\![\, s', t' \,]\!]$ is to remove states from $R(\mathbb{\Pi}_{\mathcal{H}}[\![\, p' \,]\!](\mathcal{E}_{0h}, \emptyset))$, and so, it does not invalidate (ii). Moreover, as assignments and guards do not modify the scheduler state, the subsequent *sched* application has no effect. Consider now the case where s' is a synchronization primitive. Then (i) holds as these primitives do not modify the error set. We now prove (ii). Given $(h, \rho) \in R(\mathbb{\Pi}_{\mathcal{H}}[\![\, p \,]\!](\mathcal{E}_{0h}, \emptyset))$, there is a corresponding $(h_1, \rho_1) \in R(\mathbb{\Pi}_{\mathcal{H}}[\![\, p' \,]\!](\mathcal{E}_{0h}, \emptyset))$ such that $(h, \rho) \in R(\mathbb{S}_{\mathcal{H}}[\![\, s', t' \,]\!](\{(h_1, \rho_1)\}, \emptyset))$. Given $t \in \mathcal{T}$, we apply (ii) on (h_1, ρ_1) and p', and get a state $(c_1, \rho_1') \in R(\mathbb{\Pi}_{\mathcal{C}}[\![\, \mathit{proj}_t(p'), t \,]\!](\mathcal{E}_{0c}, \emptyset, I_{\mathcal{C}}))$ with $c_1 \in \alpha_t(h_1)$. We will note $(l_1, u_1, -)$ the components of c_1. We construct a state in $\mathcal{C} \times \mathcal{E}$ satisfying (ii) for p. We first study the case where $t = t'$ and consider several sub-cases depending on s':

- Case $s' = \mathbf{yield}$. We have $\rho_1 = \rho$ and $\alpha_t(h_1) = \alpha_t(h)$. We choose $c = (l_1, \emptyset, \mathit{weak})$. Then $(c, \rho_1') \in R(\mathbb{S}_{\mathcal{C}}[\![\, s',\ t \,]\!](\{\, (c_1, \rho_1') \,\}, \Omega, I_{\mathcal{C}}))$. Moreover, $c \in \alpha_t(h)$. We also have, $\forall c' \in \mathcal{C} : \mathit{intf}(c_1, c') \Longrightarrow \mathit{intf}(c, c')$. Hence, $\forall X \in \mathcal{V}$: either $\rho(X) = \rho_1'(X)$, or $\rho_1'(X)$ comes from some weakly consistent interference not in exclusion with c_1, and so, not in exclusion with c. As a consequence, (c, ρ_1') satisfies (ii) for p'.
- Case $s' = \mathbf{lock}(m)$. We choose $c = (l_1 \cup \{m\}, \emptyset, \mathit{weak})$. This ensures as before that $c \in \alpha_t(h)$. Moreover, $\rho_1 = \rho$. We now construct ρ' such that

  $$(c, \rho') \in R(\mathbb{S}_{\mathcal{C}}[\![\, s',\ t \,]\!](\{\, (c_1, \rho_1') \,\}, \Omega, I_{\mathcal{C}}))$$

  and

  $$\forall X \in \mathcal{V} : \rho(X) = \rho'(X) \text{ or } \exists t'', c' : t' \neq t'', \mathit{intf}(c, c'),\ (t'', c', X, \rho(X)) \in I_{\mathcal{C}}\ .$$

  - If $\rho_1'(X) = \rho_1(X)$, then we take $\rho'(X) = \rho_1'(X) = \rho_1(X) = \rho(X)$.
  - Otherwise, we know that $\rho(X) = \rho_1(X)$ comes from a weakly consistent interference compatible with c_1: $\exists t'', c' : t' \neq t''$, $\mathit{intf}(c_1, c')$, $(t'', c', X, \rho_1(X)) \in I_{\mathcal{C}}$. If $\mathit{intf}(c, c')$ as well, then the same weakly consistent interference is compatible with c and can be applied to ρ'. We can thus set $\rho'(X) = \rho_1'(X)$.
  - Otherwise, as $\mathit{intf}(c, c')$ does not hold, then either $m \in l'$ or $m \in u'$, where $(l', u', -) = c'$.
    Assume that $m \in l'$, i.e., $\rho_1(X)$ was written to X by thread t'' while holding the mutex m. Because $R(\mathbb{\Pi}_{\mathcal{H}}[\![\, p \,]\!](\mathcal{E}_{0h}, \emptyset)) \neq \emptyset$, the mutex m is unlocked before t' executes $s' = \mathbf{lock}(m)$, so, t'' executes $\mathbf{unlock}(m)$ at some point in an environment mapping X to $\rho_1(X)$. Note that $\mathbb{S}_{\mathcal{C}}[\![\, \mathbf{unlock}(m),\ t'' \,]\!]$ calls *out* to convert the weakly consistent interference $(t'', c', X, \rho_1(X)) \in I_{\mathcal{C}}$ to a well synchronized interference $(t'', (l' \backslash \{m\}, u', \mathit{sync}(m)), X, \rho_1(X)) \in I_{\mathcal{C}}$. This interference is then imported by $\mathbb{S}_{\mathcal{C}}[\![\, \mathbf{lock}(m),\ t' \,]\!]$ through *in*. Thus:

    $$(c, \rho_1'[X \mapsto \rho_1(X)]) \in R(\mathbb{\Pi}_{\mathcal{C}}[\![\, \mathit{proj}_t(p),\ t \,]\!](\mathcal{E}_{0c}, \emptyset, I_{\mathcal{C}}))$$

and we can set $\rho'(X) = \rho_1(X) = \rho(X)$.
The case where $m \in u'$ is similar, except that the weakly consistent interference is converted to a well synchronized one by a statement $\mathbb{S}_{\mathcal{C}}[\![\,\mathbf{yield},\, t''\,]\!]$ or $\mathbb{S}_{\mathcal{C}}[\![\,\mathbf{unlock}(m'),\, t''\,]\!]$ for an arbitrary mutex m', in an environment where X maps to $\rho_1(X)$.

In all three cases, (c, ρ') satisfies (ii) for p'.

- Case $s' = \mathbf{unlock}(m)$. We have $\rho_1 = \rho$. We choose $c = (l_1 \setminus \{m\}, u_1, \mathit{weak})$, which implies $c \in \alpha_t(h)$. Moreover, as in the case of **yield**, $\forall c' : \mathit{intf}(c_1, c') \Longrightarrow \mathit{intf}(c, c')$. Similarly, (c, ρ'_1) satisfies (ii) for p'.
- Case $s' = X \leftarrow \mathbf{islocked}(m)$. We have $\forall Y \neq X : \rho_1(Y) = \rho(Y)$ and $\rho(X) \in \{0, 1\}$. When $X \leftarrow \mathbf{islocked}(m)$ is interpreted as $X \leftarrow [0, 1]$, the result is straightforward: we set $c = c_1$ and $\rho' = \rho'_1[X \mapsto \rho(X)]$. Otherwise, if $\rho(X) = 0$, we set $c = (l_1, u_1 \cup \{m\}, \mathit{weak})$ and, if $\rho(X) = 1$, we set $c = (l_1, u_1 \setminus \{m\}, \mathit{weak})$, so that $c \in \alpha_t(h)$. Moreover, when $\rho(X) = 0$, then ρ' is constructed as in the case of $\mathbf{lock}(m)$, except that $\rho'(X)$ is set to 0. Likewise, the case $\rho(X) = 1$ is similar to that of **yield** and $\mathbf{unlock}(m)$, except that we also set $\rho' = \rho'_1[X \mapsto 1]$. In all cases (c, ρ') satisfies (ii) for p'.

We now study the case $t \neq t'$, which implies $\mathit{proj}_t(p) = \mathit{proj}_t(p')$. We prove that, in each case, (c_1, ρ'_1) satisfies (ii) for p:

- Case $s' = \mathbf{yield}$. We have $\rho_1 = \rho$ and $\alpha_t(h_1) = \alpha_t(h)$.
- Case $s' = \mathbf{lock}(m)$. We have $\rho_1 = \rho$. In order ensure that $c_1 \in \alpha_t(h)$, we need to ensure that $m \notin u_1$. We note that, by definition of $\mathbb{S}_{\mathcal{C}}$, a configuration with $m \in u_1$ can only be reached if the control path $\mathit{proj}_t(p')$ executed by the thread t contains some $X \leftarrow \mathbf{islocked}(m)$ statement not followed by any $\mathbf{lock}(m)$ nor **yield** statement, and no thread $t'' > t$ locks m. We deduce that $t' < t$. Hence, the interleaved control path p' preempts the thread t after a non-blocking operation to schedule a lower-priority thread t'. This is forbidden by the *enabled* function: we have $R(\mathit{enabled}_{t'}(\mathbb{\Pi}_{\mathcal{H}}[\![\,p'\,]\!](\mathcal{E}_{0h}, \emptyset))) = \emptyset$, hence $R(\mathbb{\Pi}_{\mathcal{H}}[\![\,p\,]\!](\mathcal{E}_{0h}, \emptyset)) = \emptyset$.
- Case $s' = \mathbf{unlock}(m)$. We have $\rho_1 = \rho$ and $\alpha_t(h_1) \subseteq \alpha_t(h)$.
- Case $s' = X \leftarrow \mathbf{islocked}(m)$. We have $\alpha_t(h_1) = \alpha_t(h)$. Moreover, $\forall Y \neq X : \rho_1(Y) = \rho(Y)$ and $\rho(X) \in \{0, 1\}$. To prove that (ii) holds, it is sufficient to prove that $\exists t'', c' : t \neq t'', (t'', c', X, \rho(X)) \in I_{\mathcal{C}}$ and $\mathit{intf}(c_1, c')$, so that the value of $\rho(X)$ can be seen as an interference from some t'' in t. We choose $t'' = t'$. Moreover, as c', we choose the configuration obtained by applying the recurrence hypothesis (ii) to p' on (h_1, ρ_1), but t' instead of t. We deduce then by, definition of $\mathbb{S}_{\mathcal{C}}[\![\,X \leftarrow \mathbf{islocked}(m),\, t'\,]\!]$ on the configuration c', that there exist interferences $(t', c', X, 0), (t', c', X, 1) \in I_{\mathcal{C}}$.

This ends the proof that $\mathbb{P}_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$.

We now prove that $\mathbb{P}'_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$.

The proof is similar to A.7. Given an original path p and the transformed path p' of a thread t, given any (R, Ω, I) such that I is consistent with fresh and local variables, we

prove:

(i) $\Omega(\mathbb{\Pi}_{\mathcal{C}}[\![p', t]\!](R, \Omega, I)) \subseteq \Omega(\mathbb{\Pi}_{\mathcal{C}}[\![p, t]\!](R, \Omega, I))$

(ii) $\forall (t', c, X, v) \in I(\mathbb{\Pi}_{\mathcal{C}}[\![p', t]\!](R, \Omega, I)) :$
$(t', c, X, v) \in I(\mathbb{\Pi}_{\mathcal{C}}[\![p, t]\!](R, \Omega, I))$ or $t = t' \wedge X \in \mathcal{V}_l(t)$

(iii) $\forall (c, \rho') \in R(\mathbb{\Pi}_{\mathcal{C}}[\![p', t]\!](R, \Omega, I)) : \exists \rho \in \mathcal{E} : (c, \rho) \in R(\mathbb{\Pi}_{\mathcal{C}}[\![p, t]\!](R, \Omega, I)),$
$\forall X \in \mathcal{V} : \rho(X) = \rho'(X)$ or $X \in \mathcal{V}_f$

Note, in particular, that in (ii) and (iii), the configuration c is the same for the original path p and the transformed path p'.

We first consider the case of acceptable elementary operations from Def. 3.4. As they involve guards and assignments only, not synchronization primitives, and because $\mathbb{S}_{\mathcal{C}}[\![X \leftarrow e, t]\!]$ and $\mathbb{S}_{\mathcal{C}}[\![e \bowtie 0?, t]\!]$ never change the partitioning, the proof of (i)—(iii) for all elementary transformations is identical to that of proof A.7.

We now prove that, if (i)–(iii) hold for a pair p, p', then they also hold for a pair $p \cdot s$, $p' \cdot s$ for any statement s. The case where s is an assignment or guard is also identical to that of proof A.7, for the same reason. We now consider the case of synchronization primitives. (i) holds because synchronization primitives do not change the set of errors. The proof that (ii) holds is similar to the case of an assignment. Indeed, any interference added by s after p' has the form $(t, c, X, \rho'(X))$ for some state $(c, \rho') \in R(\mathbb{\Pi}_{\mathcal{C}}[\![p', t]\!](R, \Omega, I))$. Due to (iii), there is some $(c, \rho) \in R(\mathbb{\Pi}_{\mathcal{C}}[\![p, t]\!](R, \Omega, I))$ where, either $X \in \mathcal{V}_f$ or $\rho(X) = \rho'(X)$. When $X \notin \mathcal{V}_f$, we note that any $(t, c, X, \rho'(X))$ added by s after p' is also added by s after p. Thus, the extra interferences in p' do not violate (ii). For the proof of (iii), take $(c, \rho') \in R(\mathbb{\Pi}_{\mathcal{C}}[\![p' \cdot s, t]\!](R, \Omega, I))$. There exists some $(c, \rho'_1) \in R(\mathbb{\Pi}_{\mathcal{C}}[\![p', t]\!](R, \Omega, I))$ where, for all $X \in \mathcal{V}$, either $\rho'(X) = \rho'_1(X)$, or there is a well synchronized interference $(t', c', X, \rho'(X))$ with $t \neq t'$. By applying (ii) to the pair p, p', all these interferences are also in $I(\mathbb{\Pi}_{\mathcal{C}}[\![p, t]\!](R, \Omega, I))$. Thus, by applying (iii) to the pair p, p', we get a state $(c, \rho_1) \in R(\mathbb{\Pi}_{\mathcal{C}}[\![p, t]\!](R, \Omega, I))$ that also satisfies (iii) for the pair $p \cdot s$, $p' \cdot s$.

As in proof A.7, the fact that, for any p, p', p'', q, if the pairs p, p' and p', p'' satisfy (i)–(iii), then so do the pair p, p'' and the pair $q \cdot p$, $q \cdot p'$ is straightforward. This completes the proof that elementary path transformations can be applied in sequence and applied in a context containing any primitive statements (even synchronization ones) before and after the transformed part.

The end of the proof is identical to that of proof A.7. We compare the fixpoints $\mathit{lfp}\, F$ and $\mathit{lfp}\, F'$ that compute respectively the semantics of the original program $\mathbb{S}_{\mathcal{C}}[\![\mathit{body}_t, t]\!] = \mathbb{\Pi}_{\mathcal{C}}[\![\pi(\mathit{body}_t), t]\!]$ and the transformed program $\mathbb{\Pi}_{\mathcal{C}}[\![\pi'(t), t]\!]$. Then, (i) and (ii) imply that $F'(X) \sqsubseteq_{\mathcal{C}} F(X)$, and so, $\mathit{lfp}\, F' \sqsubseteq_{\mathcal{C}} \mathit{lfp}\, F$, except for interferences on local or free variables. In particular, $\Omega(\mathit{lfp}\, F') \subseteq \Omega(\mathit{lfp}\, F)$. The interference semantics of the original program contains all the errors that can occur in any program obtained by acceptable thread transformations. As a consequence, $\mathbb{P}'_{\mathcal{H}} \subseteq \mathbb{P}_{\mathcal{C}}$. □

A.9. **Proof of Theorem 4.2.** $\mathbb{P}_{\mathcal{C}} \subseteq \mathbb{P}^\sharp_{\mathcal{C}}$.

Proof. We remark that $\mathbb{P}_{\mathcal{C}}$ and $\mathbb{P}_{\mathcal{I}}$, and so, $\mathbb{P}^\sharp_{\mathcal{C}}$ and $\mathbb{P}^\sharp_{\mathcal{I}}$, are similar. In particular, the definitions for non-primitive statements and the fixpoint computation of interferences have the exact same structure. Hence, the proof A.6 applies directly to prove the soundness

of non-primitive statements as a consequence of the soundness of primitive statements. Moreover, for assignments and tests, the proof of soundness is identical to the proof A.6, but componentwise for each $c \in \mathcal{C}$. To prove the soundness of synchronization statements, we first observe the soundness of $in^\sharp$ and $out^\sharp$ in that: $\forall t \in \mathcal{T}, l, u \subseteq \mathcal{M}, m \in \mathcal{M}, V^\sharp \in \mathcal{E}^\sharp, I^\sharp \in \mathcal{I}^\sharp : \forall \rho \in \gamma_\mathcal{E}(V^\sharp)$:

$$\begin{array}{l} in(t, l, u, m, \rho, \gamma_\mathcal{I}(I^\sharp)) \subseteq \gamma_\mathcal{E}(in^\sharp(t, l, u, m, V^\sharp, I^\sharp)) \\ out(t, l, u, m, \rho, \gamma_\mathcal{I}(I^\sharp)) \subseteq \gamma_\mathcal{I}(out^\sharp(t, l, u, m, V^\sharp, I^\sharp)) \ . \end{array}$$

Secondly, we observe that $\mathbb{S}^\sharp_\mathcal{C}$ first reorganizes (without loss of information) the sets of environment tuples $R \subseteq \mathcal{C} \times \mathcal{E}$ and interference tuples $I \subseteq \mathcal{T} \times \mathcal{C} \times \mathcal{V} \times \mathbb{R}$ appearing in $\mathbb{S}_\mathcal{C}$ as functions $R' \stackrel{\text{def}}{=} \lambda c : \{ \rho \mid (c, \rho) \in R \}$ and $I' \stackrel{\text{def}}{=} \lambda t, c, X : \{ v \mid (t, c, X, v) \in I \}$. Then, it applies an abstraction in, respectively, $\mathcal{E}^\sharp$ and $\mathcal{N}^\sharp$ by replacing the set union $\cup$ by its abstract counterparts $\cup^\sharp_\mathcal{E}$ and $\cup^\sharp_\mathcal{I}$. □